Probabilistic Inference Modulo Theories

Rodrigo de Salvo Braz
SRI International

Ciaran O'Reilly
SRI International

Vibhav Gogate
U. of Texas at Dallas

Rina Dechter
U. of California, Irvine

Abstract

We present SGDPLL(T), an algorithm that solves (among many other problems) probabilistic inference modulo theories, that is, inference problems over probabilistic models defined via a logic theory provided as a parameter (currently, equalities and inequalities on discrete sorts). While many solutions to probabilistic inference over logic representations have been proposed, SGDPLL(T) is simultaneously (1) lifted, (2) exact and (3) modulo theories, that is, parameterized by a background logic theory. This offers a foundation for extending it to rich logic languages such as data structures and relational data. By lifted, we mean that our proposed algorithm can leverage first-order representations to solve some inference problems in constant or polynomial time in the domain size (the number of values that variables can take), as opposed to exponential time offered by propositional algorithms.

1 Introduction

Uncertainty representation, inference and learning are important goals in Artificial Intelligence. In the past few decades, neural networks and graphical models have made much progress towards those goals, but even today their main methods can only support very simple types of representations (such as tables and weight matrices) that exclude logical constructs such as relations, functions, arithmetic, lists and trees. Moreover, such representations require models involving discrete variables to be specified at the level of their individual values, making generic algorithms expensive for finite domains and impossible for infinite ones.

For example, consider the following conditional probability distributions, which would need to be either automatically expanded into large tables (a process called *propositionalization*), or manipulated in a manual, ad hoc manner, in order to be processed by mainstream neural networks or graphical model algorithms:

- $P(x > 10 \,|\, y \neq 98 \vee z \leq 15) = 0.1$, for $x, y, z \in \{1, \dots, 1000\}$
- $P(x \neq \mathit{Bob} \,|\, \mathit{friends}(x, \mathit{Ann})) = 0.3$

The Statistical Relational Learning [Getoor and Taskar, 2007] literature offered more expressive languages but relied on conversion to conventional representations to perform inference, which can be very inefficient. To counter this, lifted inference [Poole, 2003; de Salvo Braz, 2007] offered solutions for efficiently processing logically specified models, but with languages of limited expressivity (such as function-free ones) and algorithms that are hard to extend. More recently, probabilistic programming [Goodman *et al.*, 2012] has offered inference on full programming languages, but relies on approximate methods on the propositional level.

We present SGDPLL(T), an algorithm that solves (among many other problems) probabilistic inference on models defined over logic representations. Importantly, the algorithm is agnostic with respect to which particular logic theory is used, which is provided to it as a parameter. In this paper, we use the theory consisting of equalities over finite discrete types, and inequalities over bounded integers, as an example. However, SGDPLL(T) offers a foundation for extending it to richer theories involving relations, arithmetic, lists and trees. While many algorithms for probabilistic inference over logic representations have been proposed, SGDPLL(T) is simultaneously (1) lifted, (2) exact[1] and (3) modulo theories. By lifted, we mean that our proposed algorithm can leverage first-order representations to solve some inference problems in constant or polynomial time in the domain size (the number of values that variables can take), as opposed to exponential time for propositional algorithms.

SGDPLL(T) is a generalization of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for solving the satisfiability problem. SGDPLL(T) generalizes DPLL in three ways: (1) while DPLL only works on propositional logic, SGDPLL(T) takes (as mentioned) a logic theory as a parameter; (2) it solves many more problems than satisfiability on boolean formulas, including summations over real-typed expressions, and (3) it is *symbolic*, accepting input with free variables (which can be seen as constants with unknown values) in terms of which the output is expressed.

Generalization (1) is similar to the generalization of DPLL made by Satisfiability Modulo Theories (SMT) [Barrett *et al.*,

[1] Our emphasis on exact is not due to its being a more useful inference in practical applications, but to the idea that it is a needed basis for flexible and well-understood approximations.

2009; de Moura *et al.*, 2007; Ganzinger *et al.*, 2004], but SMT algorithms require only satisfiability solvers of their theory parameter to be provided, whereas SGDPLL(T) may require solvers for harder tasks (including model counting) that depend on the theory, including *symbolic model counters*, i.e., Figures 1 and 2 illustrate how both DPLL and SGDPLL(T) work and highlight their similarities and differences.

Note that SGDPLL(T) is not a *probabilistic* inference algorithm in a *direct* sense, because its inputs are not defined as probabilistic distributions, random variables, or any other concepts from probability theory. Instead, it is an *algebraic* algorithm defined in terms of expressions, functions, and quantifiers. However, probabilistic inference on rich languages can be directly reduced to problems that SGDPLL(T) can efficiently solve.

2 Intuition: DPLL, SMT and SGDPLL(T)

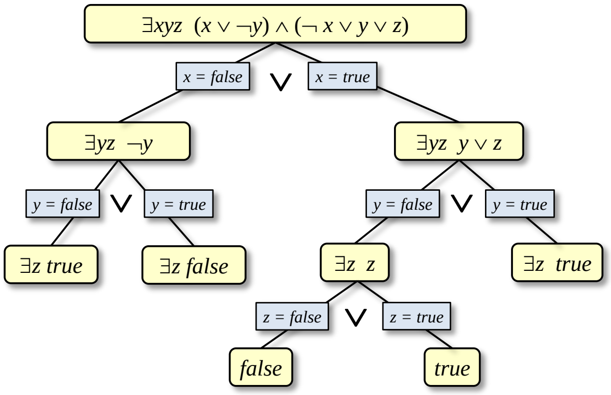
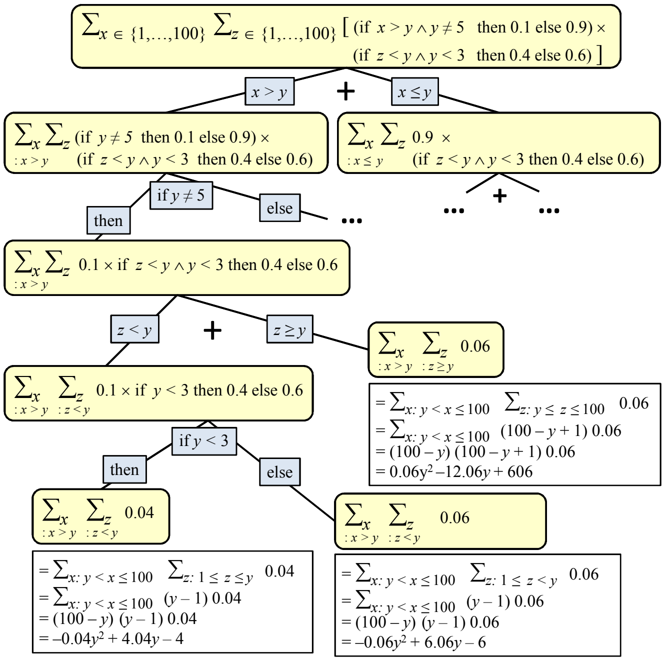


Figure 1: Example of DPLL's search tree for the existence of satisfying assignments. We show the full tree even though the search typically stops when the first satisfying assignment is found.

The **Davis-Putnam-Logemann-Loveland (DPLL)** algorithm [Davis *et al.*, 1962] solves the **satisfiability** (or **SAT**) problem. SAT consists of determining whether a propositional formula F, expressed in conjunctive normal form (CNF) has a solution or not. A CNF is a conjunction ($\wedge$) of clauses where a clause is a disjunction ($\vee$) of literals, where a literal is a proposition (e.g., x) or its negation (e.g., $\neg x$). A solution to a CNF is an assignment of values from the set $\{0, 1\}$ or $\{\text{TRUE}, \text{FALSE}\}$ to all Boolean variables (or propositions) in F such that at least one literal in each clause in F is assigned to TRUE.

Algorithm 1 shows a simplified and non-optimized version of DPLL which operates on CNF formulas. It works by recursively trying assignments for each proposition, one at a time, simplifying the CNF, and terminating if F is a constant (TRUE or FALSE). Figure 1 shows an example of the execution of DPLL. Although simple, DPLL is the basis for modern SAT solvers which improve it by adding sophisticated techniques and optimizations such as unit propagation, watch literals, and clause learning [Eén and Sörensson, 2003].

Satisfiability Modulo Theories (SMT) algorithms [Barrett *et al.*, 2009; de Moura *et al.*, 2007; Ganzinger *et al.*, 2004] generalize DPLL and can determine the satisfiability of a Boolean formula expressed in first-order logic, where some function and predicate symbols have additional interpretations. Examples of predicates include equalities, inequalities, and uninterpreted functions, which can then be evaluated using rules of real arithmetic. SMT algorithms condition on the *literals* of a background theory T, looking for a truth assignment to these literals that satisfies the formula. While a SAT solver is free to condition on a proposition, assigning it to either TRUE or FALSE regardless of previous choices (truth values of propositions are independent from each other), an SMT solver needs to also check whether a choice for one literal is *consistent* with the previous choices for others, according to T. This is done by a theory-specific model checker provided to the SMT algorithm as a parameter.



Σ x ∈ {1,…,100} Σ z ∈ {1,…,100} [(if x > y ∧ y ≠ 5 then 0.1 else 0.9) × (if z < y ∧ y < 3 then 0.4 else 0.6)]
x > y
+
x ≤ y
Σx Σz (if y ≠ 5 then 0.1 else 0.9) × (if z < y ∧ y < 3 then 0.4 else 0.6) :x > y
Σx Σz 0.9 × (if z < y ∧ y < 3 then 0.4 else 0.6) :x ≤ y
if y ≠ 5
then
else
...
...
+
...
Σx Σz 0.1 × if z < y ∧ y < 3 then 0.4 else 0.6 :x > y
z < y
+
z ≥ y
Σx Σz 0.06 :x > y :z ≥ y
= Σx: y < x ≤ 100 Σz: y ≤ z ≤ 100 0.06
= Σx: y < x ≤ 100 (100 – y + 1) 0.06
= (100 – y) (100 – y + 1) 0.06
= 0.06y² –12.06y + 606
Σx Σz 0.1 × if y < 3 then 0.4 else 0.6 :x > y :z < y
if y < 3
then
else
Σx Σz 0.04 :x > y :z < y
= Σx: y < x ≤ 100 Σz: 1 ≤ z ≤ y 0.04
= Σx: y < x ≤ 100 (y – 1) 0.04
= (100 – y) (y – 1) 0.04
= –0.04y² + 4.04y – 4
Σx Σz 0.06 :x > y :z < y
= Σx: y < x ≤ 100 Σz: 1 ≤ z < y 0.06
= Σx: y < x ≤ 100 (y – 1) 0.06
= (100 – y) (y – 1) 0.06
= –0.06y² + 6.06y – 6

Figure 2: SGDPLL(T) for summation with a background theory of inequalities on bounded integers. It splits the problem according to literals in the background theory, simplifying it until the sum is over a literal-free expression. Some of the splits are on a free variable (y) and create if-then-else expressions which are symbolic conditional solutions. Other splits are on quantified variables (x, z), and split the corresponding quantifier. When the base case with a literal-free expression is obtained, the specific theory solver computes its solution (white boxes). This figure does not show how the conditional subsolutions are summed together; see end of Section 4.1 for examples and details.

SGDPLL(T) is, like SMT algorithms, **modulo theories** but further generalizes DPLL by being **symbolic** and **quantifier-parametric** (thus "Symbolic Generalized DPLL(T)"). These three features can be observed in the prob-

Algorithm 1 A version of the DPLL algorithm.

```
DPLL($F$)
    $F$: a formula in CNF.
    $simplify$: simplifies boolean formulas given a condition
    (e.g., $simplify(x \wedge y | \neg y) = $ FALSE)

if $F$ is a boolean constant
    return $F$
else $v \leftarrow$ pick a variable in $F$
    $Sol_1 \leftarrow DPLL(simplify(F| \ \ v))$
    $Sol_2 \leftarrow DPLL(simplify(F|\neg v))$
    return $Sol_1 \vee Sol_2$
```

lem being solved by SGDPLL(T) in Figure 2:

$$\sum_{x,z \in \{1,\dots,100\}} (\texttt{if}\, x > y \wedge y \neq 5 \;\texttt{then}\, 0.1 \;\texttt{else}\, 0.9) \times (\texttt{if}\, z < y \wedge y < 3 \;\texttt{then}\, 0.4 \;\texttt{else}\, 0.6)$$

In this example, the problem being solved requires more than propositional logic theory since equality, inequality and other functions are involved. The problem's quantifier is a summation, as opposed to DPLL and SMT's existential quantification $\exists$. Also, the output will be *symbolic* in y because this variable is not being quantified, as opposed to DPLL and SMT algorithms which implicitly assume all variables to be quantified.

Before formally describing SGDPLL(T), we will briefly comment on its three key generalizations.

Quantifier-parametric Satisfiability can be seen as computing the value of an existentially quantified formula; the existential quantifier can be seen as an indexed form of disjunction, so we say it is **based** on disjunction. SGDPLL(T) generalizes SMT algorithms with respect to the problem being solved by computing expressions quantified by ***any* quantifier $\bigoplus$ based on an associative commutative operation** $\oplus$. Examples of ($\bigoplus$, $\oplus$,) pairs are ($\forall$,$\wedge$), ($\exists$,$\vee$), ($\sum$,+), and ($\prod$,$\times$). Therefore SGDPLL(T) can solve not only satisfiability (since disjunction is associative commutative), but also validity (using the $\forall$ quantifier), sums, products, model counting, weighted model counting, maximization, and many more.

Modulo Theories SMT generalizes the propositions in SAT to literals in a given theory T, but the theory connecting these literals remains that of boolean connectives. SGDPLL(T) takes a theory $T = (T_{\mathcal{C}}, T_{\mathcal{L}})$, composed of a **constraint theory** $T_{\mathcal{C}}$ and an **input theory** $T_{\mathcal{L}}$. DPLL propositions are generalized to literals in $T_{\mathcal{C}}$ in SGDPLL(T), whereas the boolean connectives are generalized to functions in $T_{\mathcal{L}}$. In the example above, $T_{\mathcal{C}}$ is the theory of inequalities on bounded integers, whereas $T_{\mathcal{L}}$ is the theory of $+$, $\times$, boolean connectives and `if then else`. Of the two, $T_{\mathcal{C}}$ is the crucial one, on which inference is performed, while $T_{\mathcal{L}}$ is used simply for the simplifications after conditioning, which takes time at most linear in the input expression size.

Symbolic Both SAT and SMT can be seen as computing the value of an existentially quantified formula in which *all* variables are quantified, and which is always equivalent to either TRUE or FALSE. SGDPLL(T) further generalizes SAT and SMT by accepting quantifications over *any subset* of the variables in its input expression (including the empty set). The non-quantified variables are **free variables**, and the result of the quantification will typically depend on them. Therefore, SGDPLL(T)'s output is a **symbolic** expression in terms of free variables. Section 3 shows an example of a symbolic solution.

Being symbolic allows SGDPLL(T) to conveniently solve a number of problems, including quantifier elimination and exploitation of factorization in probabilistic inference, as discussed in Section 5.

3 T-Problems and T-Solutions

SGDPLL(T) receives a T**-problem** (or, for short, simply a **problem**) of the form

$$\bigoplus_{x_1:C_1} \cdots \bigoplus_{x_m:C_m} E,$$

where, for each $i = 1, \dots, m$, x_i is an **index variable** quantified by $\bigoplus$ and subject to constraint C_i in $T_{\mathcal{C}}$, and E an expression in $T_{\mathcal{L}}$. C_i is assumed to be equivalent to a conjunction of literals in $T_{\mathcal{C}}$. There may be **free variables**, that is, variables that are not quantified, present in both the constraints and E. An example of a problem is

$$\sum_{y} \sum_{x:3 \leq x \wedge x \leq y} \texttt{if}\, x > 4 \;\texttt{then}\, y \;\texttt{else}\, 10 + z,$$

for x, y bounded integer variables in, say, $\{1, \dots, 20\}$.

A T**-solution** (or, for short, simply a **solution**) to a problem is a quantifier-free expression in $T_{\mathcal{L}}$ equivalent to the problem. It can be an **unconditional solution**, containing no literals in $T_{\mathcal{C}}$, or a **conditional solution** of the form $\texttt{if}\, L \;\texttt{then}\, S_1 \;\texttt{else}\, S_2$, where L is a literal in $T_{\mathcal{C}}$ and S_1, S_2 are two solutions (either conditional or unconditional). Note that, in order for the solution to be equivalent to the problem, only variables that were free (not quantified) can appear in the literal L. In other words, a solution can be seen as a decision tree on literals, with literal-free expressions in the leaves, such that each leaf is equivalent to the original problem, provided that the literals on the path to it are true. For example, the problem

$$\sum_{x:1 \leq x \wedge x \leq 10} \texttt{if}\, y > 2 \wedge w > y \;\texttt{then}\, y \;\texttt{else}\, 4$$

has an equivalent conditional solution

$$\texttt{if}\, y > 2 \;\texttt{then}\, \texttt{if}\, w > y \;\texttt{then}\, 10y \;\texttt{else}\, 40 \;\texttt{else}\, 40.$$

4 SGDPLL(T)

In this section we provide the details of SGDPLL(T), described in Algorithm 2 and exemplified in Figure 2. We first give an informal explanation guided by examples, and then provide a formal description of the algorithm.

4.1 Informal Description of SGDPLL(T)

Base Case Problems

A problem is in **base case 0** if and only if $m = 1$, E contains no literals in $T_{\mathcal{C}}$ and C is in a **base form** specific to the theory T, which its solver must be able to recognize.

In our running example, we a slight variant of **difference arithmetic** [de Moura *et al.*, 2007], with atoms of the form $x < y$ or $x \leq y + c$, where c is an integer constant. Strict inequalities $x < y + c$ can be represented as $x \leq y + c - 1$ and the negation of $x \leq y + c$ is $y \leq x - c - 1$. From now on, we shorten $a \leq x \wedge x \leq b$ to $a \leq x \leq b$.

The base case for difference arithmetic is $\sum_{x:l<x\leq u} E$, where E is a polynomial and x's lower and upper bounds l and u are either variables, or differences between a variable and an integer constant. One example is $\sum_{x:y+1\leq x\leq z-3} y^2 + 4x^3$. When $l < u$, Faulhaber's formula [Knuth, 1993] allows us to compute a new polynomial E' (in the variables other than x) equivalent to the problem (details shown in the Appendix). Moreover, this can be done (a little surprisingly) in time only dependent in the *degree* of the polynomial E, *not* on the domain size of x or the distance $u - l$. If $l < u$ is false, there are no values of x satisfying the constraint, and the problem results in 0. Therefore, the solution is the conditional $\texttt{if}\, l < u \,\texttt{then}\, E' \,\texttt{else}\, 0$.

A **base case 1** problem is such that $m > 1$ and $\bigoplus_{x:C_m} E$ satisfies base case 0, yielding solution S. In this case, we reduce the problem to the simpler

$$\bigoplus_{x_1:C_1} \cdots \bigoplus_{x_{m-1}:C_{m-1}} S.$$

Non-Base case Problems

For non-base cases, SGDPLL(T) mirrors DPLL, by selecting a **splitter literal** to split the problem on, generating two simpler problems. This eventually leads to base case problems.

The splitter literal L can come from either the expression E, to bring it closer to being literal-free, or from C_m, to bring it closer to satisfying the base form prerequisites. We will see examples shortly.

Once the splitter literal L is chosen, it splits the problem in two possible ways: if L does *not* contain any of the indices x_i, it causes an **if-splitting** in which L is the condition of an `if then else` expression and the two simpler subproblems are its *then* and *else* clauses; if L contains at least one index, it causes a **quantifier-splitting** based on the latest of the indices it contains.

For an example of an if-splitting on a literal coming from E, consider

$$\sum_{z} \sum_{x:3<x\leq 10} \texttt{if}\, y > 4 \,\texttt{then}\, y \,\texttt{else}\, 10.$$

This is not a base case because E is not literal-free. However, splitting on $y > 4$ reduces the problem to

$$\texttt{if}\, y > 4 \,\texttt{then} \sum_{z} \sum_{x:3<x\leq 10} y \,\texttt{else} \sum_{z} \sum_{x:3<x\leq 10} 10,$$

containing two base cases.

For another example of an if-splitting, but this time on a literal coming from C_m, consider the problem

$$\sum_{z} \sum_{x:y<x\wedge 3<x\wedge x\leq 10} y^2.$$

This is not a base case because the constraint includes two lower bounds for x (y and 3), which are not redundant because we do not know which one is the smallest. We can however reduce the problem to base case ones by splitting the problem according to $y < 3$:

$$\texttt{if}\, y < 3 \,\texttt{then} \sum_{z} \sum_{x:3<x\leq 10} y^2 \,\texttt{else} \sum_{z} \sum_{x:y<x\leq 10} y^2.$$

For an example of quantifier-splitting on a literal coming from E, consider this problem in which the splitter literal contains at least one index (here it contains two, x and z):

$$\sum_{x:3<x\leq 10} \sum_{z} \texttt{if}\, x > 4 \,\texttt{then}\, y \,\texttt{else}\, 10 + z.$$

In this case, we cannot simply move the literal outside the scope of the sum in its latest index x. Instead, we add the literal and its negation to the constraint on x, in two new subproblems:

$$= \Big(\sum_{x:x>4\wedge 3<x\leq 10} \sum_{z} y \Big) + \Big(\sum_{x:x\leq 4\wedge 3<x\leq 10} \sum_{z} 10 + z \Big)$$
$$= \Big(\sum_{x:4<x<10} \sum_{z} y \Big) + \Big(\sum_{x:3<x\leq 4} \sum_{z} 10 + z \Big).$$

In this example, the two sub-solutions are unconditional polynomials, and their sum results in another unconditional polynomial, which is a valid solution. However, if at least one of the sub-solutions is conditional, their direct sum is not a valid solution. In this case, we need to combine them with a distributive transformation of $\oplus$ over `if then else`:

$$S \oplus (\texttt{if}\, L \,\texttt{then}\, S_1 \,\texttt{else}\, S_2)$$
$$\equiv \texttt{if}\, L \,\texttt{then}\, S \oplus S_1 \,\texttt{else}\, S \oplus S_2,$$

proceeding recursively if any of solutions S, S_2, S_3 is also conditional. For example:

$$(\texttt{if}\, x < 4 \,\texttt{then}\, y^2 \,\texttt{else}\, z) + (\texttt{if}\, y > z \,\texttt{then}\, 3 \,\texttt{else}\, x)$$
$$\equiv \texttt{if}\, x < 4 \,\texttt{then}\, y^2 + (\texttt{if}\, y > z \,\texttt{then}\, 3 \,\texttt{else}\, x)$$
$$\texttt{else}\, z \;+ (\texttt{if}\, y > z \,\texttt{then}\, 3 \,\texttt{else}\, x)$$
$$\equiv \texttt{if}\, x < 4 \,\texttt{then}\,\texttt{if}\, y > z \,\texttt{then}\, y^2 + 3 \,\texttt{else}\, y^2 + x$$
$$\texttt{else}\,\texttt{if}\, y > z \,\texttt{then}\, z \;+ 3 \,\texttt{else}\, z \;+ x.$$

The algorithm terminates because each splitting generates sub-problems with one less literal in $\bigoplus_{x_m:C_m} E$, eventually turning it into a base case and obtaining another problem with one less quantifier. When all quantifiers are eliminated, we are left with a sum of (possibly conditional) solutions, which can be summed up into a single one.

4.2 Formal Description of SGDPLL(T)

We now present a formal version of the algorithm. We start by specifying the basic tasks the given T-solver is required to solve, and then show can we can use it to solve any T-problems.

Requirements on T solver

To be a valid input for SGDPLL(T), a T-solver S_T for theory $T = (T_{\mathcal{L}}, T_{\mathcal{C}})$ must solve two tasks:

- Given a problem $\bigoplus_{x:C} E$, S_T must be able to recognize whether C is in base form and, if so, provide a solution $base_T(\bigoplus_{x:C} E)$ for the problem.
- Given a conjunction C *not* in base form, S_T must provide a tuple $split_T(C) = (L, C_L, C_{\neg L})$ such that $L \in T_{\mathcal{C}}$, and conjunctions C_L and $C_{\neg L}$ are smaller than C and satisfy $L \Rightarrow (C \Leftrightarrow C_L) \wedge \neg L \Rightarrow (C \Leftrightarrow C_{\neg L})$.

The algorithm is presented in Figure 2. Note that it does *not* depend on difference arithmetic theory, but can use a **solver for any theory satisfying the requirements above**.

If the T-solver implements the operations above in polynomial time in the number of variables and constant time in the domain size (the size of their types), then SGDPLL(T), like DPLL, will have **time complexity** exponential in the number of literals and, therefore, in the number of variables, and be **independent of the domain size**. This is the case for the solver for difference arithmetic and will be typically the case for many other solvers.

Optimizations

The algorithm as presented so far has not dealt with **redundancy** and simplification, and may generate solutions such as $\texttt{if}\ x = 3\ \texttt{then}\ \texttt{if}\ x \neq 4\ \texttt{then}\ y\ \texttt{else}\ z\ \texttt{else}\ w$ in which literals are implied (or negated) by the context they are in.

Such solutions are longer and therefore more expensive to use, and also more expensive to compute, since the redundant literal will be processed by SGDPLL(T) as usual even though that particular branch of the search tree could be pruned.

As with SMT algorithms, theory solvers can be used to detect unsatisfiable conjunctions of constraints and **prune the search** as soon as possible. This can be done by keeping a theory-specific, efficient representation of the conjunction of literals assumed so far, that can be incrementally updated in an efficient manner.

Modern SAT solvers benefit enormously from **unit propagation** and **watched literals**. In DPLL, unit propagation is performed when all but one literal L in a clause are assigned FALSE. For this **unit clause**, and as a consequence, for the CNF problem, to be satisfied, L must be TRUE and is therefore immediately assigned that value wherever it occurs, without the need to split on it. Detecting unit clauses, however, is expensive if performed by naively checking all clauses at every splitting. Watched literals is a data structure scheme that allows only a small portion of the literals to be checked instead. In the SGDPLL(T) setting, unit propagation and watched literals need to be generalized to its not-necessarily-Boolean expressions; we leave this presentation for future work.

5 Probabilistic Inference Modulo Theories

Let $P(X_1 = x_1, \ldots, X_n = x_n)$ be the joint probability distribution on random variables $\{X_1, \ldots, X_n\}$. For any tuple of indices t, we define X_t to be the tuple of variables indexed by the indices in t, and abbreviate the assignments $(X = x)$

Algorithm 2 Symbolic Generalized DPLL (SGDPLL(T)), omitting pruning, heuristics and optimizations.

```
SGDPLL($T$)($\bigoplus_{x_1:C_1} \cdots \bigoplus_{x_m:C_m} E$)
    Returns a $T$-solution for the given $T$-problem.

if $split(\bigoplus_{x_m:C_m} E)$ indicates “base case”
    $S \leftarrow base_T(\bigoplus_{x_m:C_m} E)$
    if $m = 1$ // decide if base case 0 or 1
        return $S$
    else
        $P \leftarrow \bigoplus_{x_1:C_1} \cdots \bigoplus_{x_{m-1}:C_{m-1}} S$
else
    // $split$ returned $(L, \bigoplus_{x_m:C'_m} E', \bigoplus_{x_m:C''_m} E'')$
    if $L$ does not contain any indices
        $splittingType \leftarrow$ “if”
        $Sub_1 \leftarrow \bigoplus_{x_1:C_1} \cdots \bigoplus_{x_m:C'_m} E'$
        $Sub_2 \leftarrow \bigoplus_{x_1:C_1} \cdots \bigoplus_{x_m:C''_m} E''$
    else // $L$ contains a latest index $x_j$:
        $splittingType \leftarrow$ “quantifier”
        $Sub_1 \leftarrow \bigoplus_{x_1:C_1} \cdots \bigoplus_{x_j:C_j \wedge L} \cdots \bigoplus_{x_m:C'_m} E'$
        $Sub_2 \leftarrow \bigoplus_{x_1:C_1} \cdots \bigoplus_{x_j:C_j \wedge \neg L} \cdots \bigoplus_{x_m:C''_m} E''$
    $S_1 \leftarrow$ SGDPLL($T$)($Sub_1$)
    $S_2 \leftarrow$ SGDPLL($T$)($Sub_2$)
    if $splittingType$ == “if”
        return the expression if $L$ then $S_1$ else $S_2$
    else return $combine(S_1, S_2)$

SPLIT($\bigoplus_{x:C} E$)
if $E$ contains a literal $L$
    $E' \leftarrow E$ with $L$ replaced by TRUE
    $E'' \leftarrow E$ with $L$ replaced by FALSE
    return $(L,\ \bigoplus_C E',\ \bigoplus_C E'')$
elseif $C$ is not recognized as base form by the $T$-solver
    $(L, C', C'') \leftarrow split_T(C)$
    return $(L,\ \bigoplus_{C'} E,\ \bigoplus_{C''} E)$
else return “base case”

COMBINE($S_1, S_2$)
if $S_1$ is of the form if $C_1$ then $S_{11}$ else $S_{12}$
    return the following if-then-else expression:
    if $C_1$
        then $combine(S_{11}, S_2)$
        else $combine(S_{12}, S_2)$
elseif $S_2$ is of the form if $C_2$ then $S_{21}$ else $S_{22}$
    return the following if-then-else expression:
    if $C_2$
        then $combine(S_1, S_{21})$
        else $combine(S_1, S_{22})$
else return $S_1 \oplus S_2$
```

and $(X_t = x_t)$ by simply x and x_t, respectively. Let $\bar{t}$ be the tuple of indices in $\{1, \ldots, n\}$ but not in t.

The **marginal probability distribution** of a subset of variables X_q is one of the most basic tasks in probabilistic inference, defined as

$$P(x_q) = \sum_{x_{\bar{q}}} P(x)$$

which is a summation on a subset of variables occurring in an input expression, and therefore solvable by SGDPLL(T).

If $P(X = x)$ is expressed in the language of input and constraint theories appropriate for SGDPLL(T) (such as the one shown in Figure 2), then it can be solved by SGDPLL(T), *without* first converting its representation to a much larger one based on tables. The output will be a summation-free expression in the assignment variables x_q representing the marginal probability distribution of X_q.

Other probabilistic inference problems can be equally solved by SGDPLL(T). **Belief updating** consists of computing the **posterior probability** of X_q given evidence on X_e, which is defined as

$$P(x_q|x_e) = \frac{P(x_q, x_e)}{P(x_e)} = \frac{\sum_{x_{\bar{q},\bar{e}}} P(x)}{\sum_{x_{\bar{e}}} P(x)} = \frac{P(x_q, x_e)}{\sum_{x_q} P(x_q, x_e)}$$

which be computed with two applications of SGDPLL(T), one for the marginal $P(x_q, x_e)$ and another for $P(x_e)$. [2]

We can also use SGDPLL(T) to compute the **most likely assignment** on X_q, defined by $\max_{x_q} P(x)$, since $\max$ is an associative commutative operation.

Applying SGDPLL(T) in the manner above does not take **advantage of factorized representations** of joint probability distributions, a crucial aspect of efficient probabilistic inference. However, it can be used as a basis for an algorithm, Symbolic Generalized Variable Elimination modulo theories **(SGVE(T))**, analogous to Variable Elimination (VE) [Zhang and Poole, 1994] for graphical models, that exploits factorization. Suppose $P(x)$ is represented as a product of real-valued functions (called **factors**) f_i:

$$P(x) = f_1(x_{t1}) \times \cdots \times f_m(x_{t1})$$

and we want to compute a summation over it:

$$\sum_{x_{\bar{q}}} f_1(x_{t1}) \times \cdots \times f_m(x_{tm})$$

where q and t_i are tuples.

We now choose a variable x_i for $i \notin q$ to eliminate first. Let g be the product of all factors in which x_i does not appear, h be the product of all factors in which x_i does appear, and b be the tuple of indices of variables other than x_i appearing in h. Then we rewrite the above as

$$\sum_{x_{\bar{q},\bar{i}}} g(x_{\bar{i}}) \sum_{x_i} h(x_i, x_b) = \sum_{x_{\bar{q},\bar{i}}} g(x_{\bar{i}}) h'(x_b)$$

where h' is a summation-free factor computed by SGDPLL(T) and equivalent to the innermost summation. We now have a problem of the same type as originally, but with one less variable, and can proceed until all variables in $x_{\bar{q}}$ are eliminated. The fact that SGDPLL(T) is symbolic allows us to compute h' without iterating over all values to x_i.

6 Evaluation

We did a very preliminary comparison of SGDPLL(T)-based SGVE(T), using an implementation of an equality theory ($=, \neq$ literals only) symbolic model counter, to the state-of-the-art probabilistic inference solver variable elimination and conditioning (VEC) [Gogate and Dechter, 2011], on randomly generated probabilistic graphical models based on equalities formulas, on a Intel Core i7 notebook.

We ran both SGVE(T) and VEC on a random graphical model with 10 random variables, 5 factors, with formulas of depth and breadth (number of arguments per sub-formula) 2 for random connectives $\vee$ and $\wedge$. SGVE(T) took 1.5 seconds to compute marginals for all variables (unsurprisingly, irrespective of domain size). We grounded this model for domain size 16 to provide the table-based input required by VEC, which then took 30 seconds to compute all marginals. The largest grounded table given to VEC as input had 6 random variables and therefore around 16 million entries. We did not run VEC on larger domain sizes because generating its table-based input became unmanageable.

Therefore, SGDPLL(T)-based SGVE(T) outperforms VEC even for a problem with a relatively small domain size, in spite of the fact that VEC uses modern SAT solver techniques [Marić, 2009] that are still to be incorporated into our implementation of SGDPLL(T).

We also ran SGVE(T) on many other random graphical models involving formulas with depth up to 4, breadth up to 16, and number of random variables up to 12, always computing all marginals under 3 seconds. Depths of 6 with 12 variables, however, greatly increased runtime to 160 seconds. For large formulas, a version of unit propagation and watched literals will be needed.

7 Related work

SGDPLL(T) is related to many different topics in both logic and probabilistic inference literature, besides the strong links to SAT and SMT solvers.

SGDPLL(T) is a lifted inference algorithm [Poole, 2003; de Salvo Braz, 2007; Gogate and Domingos, 2011], but the proposed lifted algorithms so far have concerned themselves only with relational formulas with equality. We have not yet developed the theory solvers for relational representations required for SGDPLL(T) to do the same, but we intend to do so using the already developed modulo-theories mechanism available. On the other hand, we have not yet developed a relational solver for SGDPLL(T) but presented one or inequalities.

SGDPLL(T) is very similar to the algorithm presented in [Belle *et al.*, 2015a] and [Belle *et al.*, 2015b], which also uses a SMT solver to split the problem to a base case. In that

[2] However, computing $P(x_q, x_e)$ and $P(x_e)$ separately requires summing $x_{\bar{q},\bar{e}}$ out of $P(x)$ and then summing $x_{q,\bar{q},\bar{e}}$ out of $P(x)$, thus summing $x_{\bar{q},\bar{e}}$ out twice. This can be avoided by using the result of the first application, the summation-free $f(x_q, x_e)$, and the fact that $P(x_e) = \sum_{x_q} P(x_q, x_e) = \sum_{x_q} f(x_q, x_e)$, thus limiting the second application to summing over x_q only.

work, the base case is weighted model integration on continuous variables, whereas ours is a sum over bounded integer variables. Another difference that we allow free variables and give them a symbolic treatment.

SGDPLL(T) generalizes several algorithms that operate on mixed networks [Mateescu and Dechter, 2008] – a framework that combines Bayesian networks with constraint networks, but with a much richer representation. By operating on richer languages, SGDPLL(T) also generalizes exact model counting approaches such as RELSAT [Bayardo, Jr. and Pehoushek, 2000] and Cachet [Sang *et al.*, 2005], as well as weighted model counting algorithms such as ACE [Chavira and Darwiche, 2008] and formula-based inference [Gogate and Domingos, 2010], which use the CNF and weighted CNF representations respectively.

8 Conclusion and Future Work

We have presented SGDPLL(T) and its derivation SGVE(T), algorithms formally able to solve a variety of problems, including probabilistic inference, modulo theories, that is, capable of being extended with theories for richer representations than propositional logic, in a lifted and exact manner.

Future work includes additional theories of interest mainly among them those for uninterpreted relations (particularly multi-arity functions) and arithmetic; modern SAT solver optimization techniques such as watched literals and unit propagation, and anytime approximation schemes that offer guaranteed bounds on approximations that converge to the exact solution.

Appendix: Summing polynomials with Faulhaber's formula

The base case 0 for summation with difference arithmetic over polynomials requires the computation of a polynomial equivalent to

$$\sum_{x: l < x \leq u} t_0 + t_1 x + \cdots + t_n x^n$$

where x is an integer index and t_i are monomials, possibly including numeric constants and powers of free variables.

Faulhaber's formula [Knuth, 1993] solves the simpler sum of powers problem $\sum_{k=1}^{n} k^p$:

$$\sum_{k=1}^{n} k^p = \frac{1}{p+1} \sum_{j=0}^{p} (-1)^j \binom{p+1}{j} B_j n^{p+1-j},$$

where B_j is a *Bernoulli number* defined as

$$\begin{aligned} B_j &= 1 - \sum_{k=0}^{j-1} \binom{j}{k} \frac{B_k}{j-k+1} \\ B_0 &= 1. \end{aligned}$$

The original problem can be reduced to a sum of powers in the following manner, where t, r, s, v, w are families of monomials (possibly including numeric constants) in the free variables:

$$\begin{aligned}
&\sum_{x: l < x \leq u} t_0 + t_1 x + \cdots + t_n x^n \\
&= \sum_{i=0}^{n} \sum_{x: l < x \leq u} t_i x^i \\
&= \sum_{i=0}^{n} \sum_{x=1}^{u-l} t_i (x+l)^i \\
&= \sum_{i=0}^{n} \sum_{x=1}^{u-l} t_i \sum_{q=0}^{i} r_q x^q \quad \text{(by expanding the binomial)} \\
&= \sum_{i=0}^{n} \sum_{x=1}^{u-l} \sum_{q=0}^{i} t_i r_q x^q \\
&= \sum_{i=0}^{n} \sum_{q=0}^{i} t_i r_q \sum_{x=1}^{u-l} x^q \text{ (inverting summations to apply Faulhaber's)} \\
&= \sum_{i=0}^{n} \sum_{q=0}^{i} \frac{t_i r_q}{q+1} \sum_{j=0}^{q} (-1)^j \binom{q+1}{j} B_j (u-l)^{q+1-j} \\
&= \sum_{i=0}^{n} \sum_{q=0}^{i} \sum_{j=0}^{q} s_{i,q,j} (u-l)^{q+1-j} \\
&= \sum_{i=0}^{n} \sum_{q=0}^{i} \sum_{j=0}^{q} s_{i,q,j} \sum_{l=1}^{q+1} v_l \quad \text{(by expanding the binomial)} \\
&= \sum_{i=0}^{n} \sum_{q=0}^{i} \sum_{j=0}^{q} \sum_{l=1}^{q+1} s_{i,q,j} v_l \\
&= w_0 + w_1 + \cdots + w_{n'} \quad \text{(since } n \text{ is a known integer constant)}
\end{aligned}$$

where n' is function of n in $O(n^4)$ (the time complexity for computing Bernoulli numbers up to B_n is in $O(n^2)$).

Because the time and space complexity of the above computation depends on the initial degree n and the degrees of free variables in the monomials, it is important to understand how these degrees are affected. Let d_l be the initial degree of the variable present in l in t monomials. Its degree is up to n in r monomials (because of the binomial expansion with i being up to n), and thus up to $d_l + n$ in s monomials (because of the multiplication of t_i and r_q). The variable has degree up to $n + 1$ in monomials v, with degree up to $d_l + 2n + 1$ in the final polynomial. The variable in u keeps its initial degree d_u until it is increased by up to $n + 1$ in v, with final degree up to $d_u + n + 1$. The remaining variables keep their original degrees. This means that degrees grow only linearly over multiple applications of the above. This combines with the $O(n^4)$ per-step complexity to a $O(n^5)$ overall complexity for n the maximum initial degree for any variable.

Acknowledgments

We gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) Probabilistic

Programming for Advanced Machine Learning Program under Air Force Research Laboratory (AFRL) prime contract no. FA8750-14-C-0005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the view of DARPA, AFRL, or the US government.

References

[Barrett *et al.*, 2009] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[Bayardo, Jr. and Pehoushek, 2000] R. J. Bayardo, Jr. and J. D. Pehoushek. Counting Models Using Connected Components. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 157–162, Austin, TX, 2000. AAAI Press.

[Belle *et al.*, 2015a] Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[Belle *et al.*, 2015b] Vaishak Belle, Guy Van den Broeck, and Andrea Passerini. Hashing-based approximate probabilistic inference in hybrid domains. In *Proceedings of the 31st Conference on Uncertainty in Artificial Intelligence (UAI)*, 2015.

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

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[de Moura *et al.*, 2007] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2007.

[de Salvo Braz, 2007] R. de Salvo Braz. *Lifted First-Order Probabilistic Inference*. PhD thesis, University of Illinois, Urbana-Champaign, IL, 2007.

[Eén and Sörensson, 2003] N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT Competition 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[Ganzinger *et al.*, 2004] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. *DPLL(T): Fast Decision Procedures*. 2004.

[Getoor and Taskar, 2007] L. Getoor and B. Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[Gogate and Dechter, 2011] V. Gogate and R. Dechter. SampleSearch: Importance sampling in presence of determinism. *Artificial Intelligence*, 175(2):694–729, 2011.

[Gogate and Domingos, 2010] V. Gogate and P. Domingos. Formula-Based Probabilistic Inference. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pages 210–219, 2010.

[Gogate and Domingos, 2011] V. Gogate and P. Domingos. Probabilistic Theorem Proving. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, pages 256–265. AUAI Press, 2011.

[Goodman *et al.*, 2012] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Daniel Tarlow. Church: a language for generative models. *CoRR*, abs/1206.3255, 2012.

[Knuth, 1993] Donald E. Knuth. Johann Faulhaber and Sums of Powers. *Mathematics of Computation*, 61(203):277–294, 1993.

[Marić, 2009] Filip Marić. Formalization and implementation of modern sat solvers. *Journal of Automated Reasoning*, 43(1):81–119, 2009.

[Mateescu and Dechter, 2008] R. Mateescu and R. Dechter. Mixed deterministic and probabilistic networks. *Annals of Mathematics and Artificial Intelligence*, 54(1-3):3–51, 2008.

[Poole, 2003] D. Poole. First-Order Probabilistic Inference. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 985–991, Acapulco, Mexico, 2003. Morgan Kaufmann.

[Sang *et al.*, 2005] T. Sang, P. Beame, and H. A. Kautz. Heuristics for Fast Exact Model Counting. In *Eighth International Conference on Theory and Applications of Satisfiability Testing*, pages 226–240, 2005.

[Zhang and Poole, 1994] N. Zhang and D. Poole. A simple approach to Bayesian network computations. In *Proceedings of the Tenth Biennial Canadian Artificial Intelligence Conference*, 1994.