Scaling Probabilistic Programming with First-Class Marginal-MAP[*]

John Gouwar, Steven Holtzen

Northeastern University
gouwar.j@northeastern.edu
s.holtzen@northeastern.edu

Abstract

One of the most compelling features of probabilistic programming languages (PPLs) is their ability to manipulate probability distributions as first-class objects. First-class representations of distributions empower PPLs to model important examples from psychology, game-playing, and model-based diagnosis; however, this expressivity comes at a cost: nested reasoning about distributions is often intractable even for very small models. In this paper we propose a new PPL that is designed around supporting an efficient implementation of a first-class marginal-MAP operator. Our core contribution is an amortized approach to exactly solving multiple (possibly nested and interacting) marginal-MAP queries on a single discrete-valued probabilistic program. Our approach is based on a branch-and-bound search that leverages compilation to probabilistic circuits in order to efficiently compute bounds. We show experimentally that our method can scale to very large programs with thousands of random variables and multiply-nested marginal-MAP queries.

```
1  cancer ∼ flip 0.001; cold ∼ flip 0.8;
2  if cancer {
3     headache ∼ flip 0.1; fever ∼ flip 0.02;
4  } else if cold {
5     headache ∼ flip 0.6; fever ∼ flip 0.3;
6  } else {
7     headache ∼ flip 0.01; fever ∼ flip 0.002;
8  }
9  observe(headache);
10 (cancer_diag, cold_diag) = map(cancer, cold);
11 if !cancer_diag && cancer {
12    complications ∼ flip 0.8;
13 } else {
14    complications ∼ flip 0.01;
15 }
16 return complications;
```

Figure 1: A motivating example of first-class Marginal-MAP in in the `PineaPPL` probabilistic programming language.

1 Introduction

Many important applications of reasoning under probabilistic uncertainty involve *meta-reasoning*: modeling sequential decision-making processes where agents make decisions based on the probability of events according to the model, sometimes referred to as nested reasoning. Probabilistic programming languages (PPLs) elegantly enable meta-reasoning by representing queries as first-class objects within the probabilistic model (Stuhlmüller and Goodman 2014; Mantadelis and Janssens 2011). This capability is illustrated in the example medical diagnosis program given in Figure 1, written in a new imperative PPL we call `PineaPPL`. This program models a doctor making a diagnosis about the patient's underlying set of diseases given observations of some subset of symptoms. Lines 1–8 define a simple probabilistic relationship between symptoms and diseases: the diseases (`cancer` and `cold`) each cause one of two symptoms (`headache` and `fever`) to occur with some probability. The syntax `cancer∼flip 0.001` indicates that the program variable `cancer` is a random variable that is true with probability 0.001.

The doctor must make a diagnosis of the most likely underlying disease and proceed with treating the patient. This situation is modeled directly in `PineaPPL` by using the `map` (short for *marginal maximum a-posteriori*, often simply called "MAP" in the literature) keyword on Line 10. This line performs meta-reasoning by querying the model for the most likely joint assignment to the diseases given observation of evidence (in this case, that the patient has a headache), and binds these values to the variables `cancer_diag` and `cold_diag`. The model can continue by then branching on these values in order to ultimately produce a distribution on whether the doctor's diagnosis led to any medical complications.

While eminently useful, few of today's PPLs support the convenient use of first-class meta-reasoning, primarily due to its fundamental computational intractability. Meta-reasoning is significantly computationally harder than the typical inference task for PPLs (Rainforth et al. 2018). As such, today's PPLs resort almost exclusively to approximate techniques to solve first class meta-reasoning tasks such as simulated annealing, Bayesian hill-climbing, or sampling (Tolpin, Van de Meent, and Wood 2015; Tolpin et al. 2021; Tolpin and Wood 2015; Rainforth et al. 2018). In the context of meta-reasoning, approximations have two main downsides. First, errors in approximations are com-

[*]This version of the paper is a non-archival draft uploaded on January 9, 2023.

pounded by nesting, leading to important issues in correctness (Rainforth et al. 2018). Second, approximations struggle to converge on problems with high-dimensional combinatorial constraints, which we will demonstrate in our experiments in Section 4. Ultimately, these constraints mean that today's PPLs are only capable of handling very small instances of meta-reasoning.

In this paper we tackle the challenge of scaling meta-reasoning for probabilistic programs to large practical programs. We focus on the important case of exact `map` inference for discrete-valued finite-domain probabilistic programs. First, we develop a new PPL called `PineaPPL` specifically for representing the class of programs for which we can perform exact `map`. Then, we show how to compile `PineaPPL` programs into a probabilistic circuit that efficiently represents the joint distribution over program variables, in a manner similar to `Dice` (Holtzen, Van den Broeck, and Millstein 2020), `ProbLog` (Fierens et al. 2015), and `SPPL` (Saad, Rinard, and Mansinghka 2021). We leverage this probabilistic circuit to solve multiple `map` queries on `PineaPPL` programs by designing an efficient branch-and-bound based search in a style similar to the technique developed by Huang et al. (2006) in the context of solving `map` for Bayesian networks. Finally, in Section 4 we show that `PineaPPL` can perform multiple exact `map` queries on programs with thousands of random variables.

2 Background

In this section we review the relevant existing work on nested inference for PPLs and `map`. The most common form of meta-reasoning in today's PPLs is *meta-inference* (often also called "nested inference"), which queries the program for the marginal probability of some event holding. Many languages support first-class meta-inference, including `WebPPL` (Stuhlmüller and Goodman 2014), `Anglican` (Tolpin, Van de Meent, and Wood 2015), and `Omega` (Tavares et al. 2019). Meta-inference is very computationally hard to realize, especially for programs involving continuous random variables, as it either involves complex interleavings of sampling processes or nested integrals (Rainforth et al. 2018). While nested inference remains computationally daunting for PPLs, there is an even harder nested reasoning query that is currently firmly out of reach for today's languages: marginal-MAP.

Definition 1 *Let* $\Pr(M, V, E)$ *be a joint probability distribution on sets of random variables* M*,* V*, and* E*. We call* M *the* MAP-variables*,* V *the* marginal variables*, and* E *the* evidence variables*. Then, the* marginal-MAP *query on* Pr *is defined for some fixed evidence* $e \in E$*:*

$$\arg\max_{m \in M} \sum_{v \in V} \Pr(M = m, V = v \mid E = e). \quad (1)$$

Marginal-MAP is provably harder than nested inference because the sequential max and marginalization cannot be commuted, placing burdens on the order in which terms can be eliminated: variables must be summed before they can be maxed. Park and Darwiche (2003) showed the consequences of this: for the case of discrete Bayesian networks, the marginal-MAP query is exponential in a constrained tree-width of the network, which is significantly harder than the usual worst-case inference complexity of tree-width.

Algorithm 1: `map`: Marginal-MAP branch/bound

```
   Data: Distribution $\Pr(M, V \mid E = e)$; lower-bound
         $l$; upper-bound function ub
   Result: A marginal-MAP assignment
1  if $M$ is empty then
2      return $\max(\sum_{v \in V} \Pr(V = v \mid E = e),\ l)$
3  else
4      $[h :: M'] \leftarrow M$;
5      for $v \in \{T, F\}$ do
6          Label $\Pr(M', V \mid E = e, h = v)$ as $\Pr'$;
7          $u \leftarrow \mathrm{ub}(\Pr')$;
8          if $u > l$ then
9              $l \leftarrow \max\big(\mathrm{map}(\Pr', l),\ l\big)$;
10         end
11     end
12     return $l$
13 end
```

To tackle the complexity of marginal-MAP, Park and Darwiche (2003) introduced a generic branch-and-bound style search, described in Algorithm 1. This algorithm takes as input (1) a distribution; (2) a lower-bound on the MAP probability (which can be trivially initialized to 0 or via an incomplete search); and (3) a function `ub` that computes an upper-bound on the MAP probability. The complexity of Algorithm 1 is governed by (1) how efficient `ub` is to evaluate; (2) how efficient the marginalization on Line 2 can be performed; and (3) the quality of the upper-bounds, which dictates the degree of pruning that can be exploited on Line 8.

Marginalizing random variables and computing a high-quality `ub` is hard for discrete Bayesian networks and probabilistic programs (Darwiche 2009; Holtzen, Van den Broeck, and Millstein 2020). To address this, Huang et al. (2006) introduced the idea of *marginal-MAP via compilation*: first, compile the distribution into a *probabilistic circuit* that supports fast (linear-time) marginalization and a high-quality `ub` function based on relaxing the marginal-MAP query to freely commute sums and products. Marginal-MAP via compilation is still provably computationally hard, even after disregarding the high cost of compilation itself (De Campos 2011). Nonetheless, empirically it remains the state-of-the-art method for solving marginal-MAP exactly for discrete graphical models (Mei, Jiang, and Tu 2018; Choi, Friedman, and Van den Broeck 2022).

3 MAP-via-Compilation for PPLs

This section introduces our core approach and contribution. Our goal is to apply the principle of marginal-MAP via compilation to a discrete-valued PPL. To do this, we will first develop a new PPL called `PineaPPL` and a compilation strategy for `PineaPPL` that enables (1) efficient computation of marginal probabilities and (2) efficient computation of high-quality upper-bounds of marginal-MAP queries.

```
e ::= x | !e | e₁ && e₂ | e₁ || e₂
s ::= x ∼ flip θ | [id] = e
      | if e { s₁ } else { s₂ } | s₁; s₂
      | (x₁ ⋯ xᵢ) = map(y₁ ⋯ yᵢ)
q ::= Pr(e) | margmap [x₁ ⋯ xᵢ]
p ::= s; [observe(e)]*; return [q₁⋯qₙ]
```

Figure 2: Core PineaPPL syntax.

```
category Language = English
                   | French
                   | Dutch ;
x ∼ sample Language ;
return [x]
```

(a) A PineaPPL program with extended categorical syntax.

```
x@English ∼ flip 0.3333 ;
if x@English {
  x@French = false ;
}
else {
  // Pr(x is French | x is not English)
  x@French ∼ flip 0.5;
}
if (x@English||x@French) {
  x@Dutch = false;
}
}
else {
  x@Dutch = true;
}
return [Pr(x@Dutch)]
```

(b) Desugaring of the previous program

Figure 3: An example of extended categorical syntax in PineaPPL and its desugaring.

3.1 Syntax

A natural question one might ask here is: *why is it necessary to develop a new language and syntax?* We have two main reasons: (1) none of today's PPLs have a formal syntax and semantics for first-class map; and (2) in order to aid in our eventual compilation strategy we need to careful restrict the class of programs that can be constructed; this will be discussed in further detail in Section 3.3.

At its core, PineaPPL is an imperative language for discrete-valued loop-free probabilistic programs with support for categorical and Boolean random variables. The core syntax of PineaPPL is given in Figure 2. A program is composed of a statement (or sequence of statements) describing a probabilistic model, a series of expressions observed to be true about the model, and a series of inference queries about the model. PineaPPL is equipped with a single probabilistic statement x ∼ flip θ, where θ is a real number between 0 and 1, denoting that x is sampled from a Bernoulli distribution with parameter θ. In addition to binding variables to probabilistic samples, variables can be assigned to Boolean expressions on previously defined variables. The final binding form in PineaPPL uses the map keyword applied to a previously defined set of variables to bind another set of variables to the the marginal-MAP (*i.e.,* the most likely assignment) of the scrutinized variables. The if-else statement allows for the conditional definition of variables guarded by the value of a Boolean expression; any variable bound in one branch must be bound in the other. Finally, while variables can be redefined, it can only be done at different nesting levels (i.e., one cannot define the same variable twice in the same branch of an if-else statement, unless guarded by a more deeply-nested if-else statement).

We extend the core syntax of PineaPPL with syntactic sugar for sampling from user-defined discrete distributions. A categorical variable can be sampled from a predefined set of variants, either uniformly at random or with explicitly defined priors (which must sum to one). We also introduce a Boolean predicate, is, which tests whether a given variable takes on the value of a particular variant. Desugaring a sample from a categorical distribution involves introducing a sequence of indicator variables each representing a variant the original variable can take on; this is similar to encoding of categoricals in other discrete-valued languages like Dice (Holtzen, Van den Broeck, and Millstein 2020) and ProbLog (Fierens et al. 2015). The first indicator variable in the sequence is represented as a flip with the defined prior, and each subsequent indicator is conditioned on the fact that none of the previous indicators are true, otherwise the indicator is set to false; this is a common encoding of categorical variables with Boolean variables known as a *one-hot encoding*. The is expression is then naturally desugared into the appropriate indicator for the test. An example of this desugaring is presented in Figure 3.

3.2 Denotational Semantics

The semantics of PineaPPL are a variant of a standard measure-transformer with the addition of the map keyword (Borgström et al. 2011). A program environment $\rho \in \texttt{Env}$ is a map from variables to values. Let $\texttt{Dist}(\texttt{Env})$ denote a distribution on all possible environments. Then, the semantics of a statement is a map between environment distributions: $[\![\texttt{s}]\!] : \texttt{Dist}(\texttt{Env}) \to \texttt{Dist}(\texttt{Env})$, and the semantics of expressions are measurable maps $[\![\texttt{e}]\!] : \texttt{Env} \to \texttt{Env}$. For example:

$$[\![\texttt{s}_1; \texttt{s}_2]\!](\mu) = [\![\texttt{s}_2]\!]\Big([\![\texttt{s}_1]\!](\mu)\Big) \tag{2}$$

These semantics are standard, except for the new map statement. For simplicity, consider a map statement that binds a single value, $[\![x_1 = \texttt{map}(y_1)]\!](\mu)$. These semantics will (1) query μ for the most likely configuration of the y_1; and (2) output a new state that assigns probability 1 the state that binds x_1 to the map value for y_1. Formally, we can write this in terms of the semantics of assignment:

$$\begin{aligned}[\![x_1 = \texttt{map}(y_1)]\!](\mu) &= [\![x = v^\star]\!](\mu) \\ \text{where } v^\star &= \arg\max_{v \in \{\texttt{T},\texttt{F}\}} \sum_{\{\rho \in \texttt{Env} \mid \rho(y)=v\}} \mu(\rho).\end{aligned} \tag{3}$$

These semantics generalize straightforwardly to the case when more than one variable is being optimized over.

$$\frac{\texttt{e} \Downarrow \varphi}{\texttt{[id] = e} \Downarrow [id \Leftrightarrow \varphi]}\ (\text{C1}) \qquad \frac{\texttt{fresh}\ f_1}{\texttt{x}{\sim}\texttt{flip}\ \theta \Downarrow [x \Leftrightarrow f_1]}\ (\text{C2})$$

$$\frac{}{\texttt{(x}_i\texttt{) = map(y}_i\texttt{)} \Downarrow [x_i \Leftrightarrow y_i]}\ (\text{C3}) \qquad \frac{s_1 \Downarrow l_1;\ s_2 \Downarrow l_2}{[l_1; l_2]}\ (\text{C4})$$

$$\frac{s_1 \Downarrow [x_i \Leftrightarrow \varphi_i];\ s_2 \Downarrow [x_i \Leftrightarrow \psi_i];\ c \Downarrow \chi}{[x_i \Leftrightarrow (\chi \wedge \varphi_i) \vee (\neg\chi \wedge \psi_i)]}\ (\text{C5})$$

$$\frac{\texttt{e}_i \Downarrow \varphi_i}{\texttt{[observe(e}_i\texttt{)]} \Downarrow \bigwedge_i [\varphi_i]}\ (\text{C6}) \qquad \frac{\texttt{e} \Downarrow \varphi}{\texttt{Pr(e)} \Downarrow \varphi}\ (\text{C7})$$

$$\frac{}{\texttt{margmap[x}_1\ \texttt{...}\ \texttt{x}_i\texttt{]} \Downarrow \top}\ (\text{C8})$$

$$\frac{\texttt{s} \Downarrow [\varphi_i];\ \texttt{[observe(e}_i\texttt{)]} \Downarrow \varphi_o;\ \texttt{[q}_i\texttt{]} \Downarrow \varphi_q}{\texttt{s; [observe(e}_i\texttt{)]; return[q}_i\texttt{]} \Downarrow (\bigwedge_i \varphi_i) \wedge \varphi_o \wedge \varphi_q}\ (\text{C9})$$

Figure 4: Complete `PineaPPL` operational semantics.

```
1   x ~ flip₁ 0.1;
2   if x {
3       y ~ flip₂ 0.2;
4   } else {
5       y ~ flip₃ 0.3;
6   }
7   if y {
8       z ~ flip₄ 0.4;
9   } else {
10      z ~ flip₅ 0.6;
11  }
12  return margmap(y);
```

(a) Simple `PineaPPL` program.

Figure 5: Example `PineaPPL` program.

3.3 Compiling PineaPPL to Boolean Formulae

While the semantics in the previous section give us a description of how `PineaPPL` programs are associated with particular probability distributions, these rules themselves are not efficiently implementable. We now describe how to compile `PineaPPL` into a representation that affords (1) efficient marginal inference; and (2) efficient computation of high-quality upper-bounds on `map` queries. We adopt an approach that is similar to other recently developed languages that work via compilation to probabilistic circuits like `Dice` (Holtzen, Van den Broeck, and Millstein 2020), `SPPL` (Saad, Rinard, and Mansinghka 2021), and `ProbLog` (Fierens et al. 2015). However, unlike these existing languages, `PineaPPL` is an imperative language with support for first-class `map`, necessitating a novel compilation strategy.

Similar to existing compiled PPLs, `PineaPPL` programs are compiled to weighted Boolean formulae (WBF), which are then translated to probabilistic circuits, such as binary decision diagrams (BDDs); this enables fast inference via the well-known *inference via weighted model counting* approach (Chavira and Darwiche 2008; Sang, Beame, and Kautz 2005). Formally, let φ be a Boolean formula and $w : L \to \mathbb{R}$ be a *weight function* that assigns real-valued weights to each literal (assignment to Boolean variable). The tuple (φ, w) is called a *weighted Boolean formula* (WBF), and it is often interpreted as a distribution via a weighted model count (`WMC`):

Definition 2 *Let* (φ, w) *be a Boolean formula. Then the* weighted model count*, denoted* $\texttt{WMC}(\varphi, w)$ *is defined:*

$$\texttt{WMC}(\varphi, w) = \sum_{m \models \varphi} \prod_{l \in m} w(l). \tag{4}$$

As in `Dice` and `ProbLog`, our goal now is to establish that the `WMC` of a compiled `PineaPPL` program corresponds with the `PineaPPL` measure-transformer semantics. To accomplish this we define a *compilation relation* $\texttt{s} \Downarrow \varphi$ that relates a `PineaPPL` syntactic statement `s` with a Boolean formula φ. Similarly, we will need a relation that associates program environments ρ with logical expressions, also defined $\rho \Downarrow \alpha$. The complete set of rules is given in Figure 4. Ultimately, these rules must satisfy a correctness criteria that relates the semantics and `WMC`:

Theorem 1 *Let* `s` *be a* `PineaPPL` *statement and assume* $p \Downarrow [\varphi_i]$*, and let* w *be the weight function. Then, for any pair of environments* ρ *and* ρ' *such that* $\rho \Downarrow \alpha$ *and* $\rho' \Downarrow \alpha'$*, we have that* $\texttt{WMC}(\bigwedge_i \varphi_i \wedge \alpha \wedge \alpha', w) = [\![s]\!](\rho)(\rho')$.

This theorem is best illustrated via an example. Consider the simple program in Figure 5. Compiling the `flip` statement on line 1 according to rule (C2) yields the single bi-implication $[\texttt{x} \Leftrightarrow \texttt{f}_1]$. The logical variable x is an *indicator variable*: it is true if and only if the *parameter variable* f_1 is true. The weight function w gives a weight of 1 to both assignments of all indicator variables, and the weight of parameter variables follows from the program: $w(\texttt{f}_1 = 0.1), w(\overline{\texttt{f}_1}) = 0.9$ (note that we use the overline notation to denote a logically negated variable). To check Theorem 1 on this example, let's consider the empty input environment $\rho = \emptyset$ and the output environment $\rho' = [x \mapsto \texttt{F}]$. Then, $\rho \Downarrow \texttt{T}$ and $\rho' \Downarrow \overline{x}$. Then, $\texttt{WMC}(x \Leftrightarrow \texttt{f}_1 \wedge \texttt{T} \wedge \overline{x}, w) = 0.9$, as expected.

Continuing to compile the entire program, we consider the interesting case of compiling the `if-else` statement on lines 2–6. This first requires applying rule (C2) again to the `flip` statements on lines 3 and 5 to get the bimplications $\texttt{y} \Leftrightarrow \texttt{f}_2$ and $\texttt{y} \Leftrightarrow \texttt{f}_3$. We then apply rule (C5) to get the single formula $\texttt{y} \Leftrightarrow ((\texttt{x} \wedge \texttt{f}_2) \vee (\neg\texttt{x} \wedge \texttt{f}_3))$. We compile the `if-else` on lines 7–11 in exactly the same way to get the formula $\texttt{z} \Leftrightarrow ((\texttt{y} \wedge \texttt{f}_4) \vee (\neg\texttt{y} \wedge \texttt{f}_5))$. Rule (C8) implies that compiling marginal-MAP queries does not add any variables to the compiled formula. Finally, we apply rule (C9) to get the final formula:

$$\begin{gathered}(\texttt{x} \Leftrightarrow \texttt{f}_1) \wedge \texttt{y} \Leftrightarrow ((\texttt{x} \wedge \texttt{f}_2) \vee (\neg\texttt{x} \wedge \texttt{f}_3)) \\ \wedge\, \texttt{z} \Leftrightarrow ((\texttt{y} \wedge \texttt{f}_4) \vee (\neg\texttt{y} \wedge \texttt{f}_5))\end{gathered} \tag{5}$$

Each compilation rule also creates a mapping from a variable and it's negation to a probability. Simple assignment, `x = e`, maps both `x` and it's negation to 1.0. Sampling from a Bernoulli distribution, $\texttt{x} \sim \texttt{flip}\ \theta$ again maps `x` and it's negation to 1.0, while maping the generated variable `f` to θ and it's negation to $1-\theta$. The weight of the variables bound by a `map` statement are contingent on the result of the most likely assignment of their corresponding scrutinized variable. If the most likely assignment is `true`, then the bound variable is given a weight of 1.0 and its negation is given a weight of 0.0; otherwise, if the most likely assignment is `false`, the bound variable is given a weight of 0.0 and its negation 1.0.

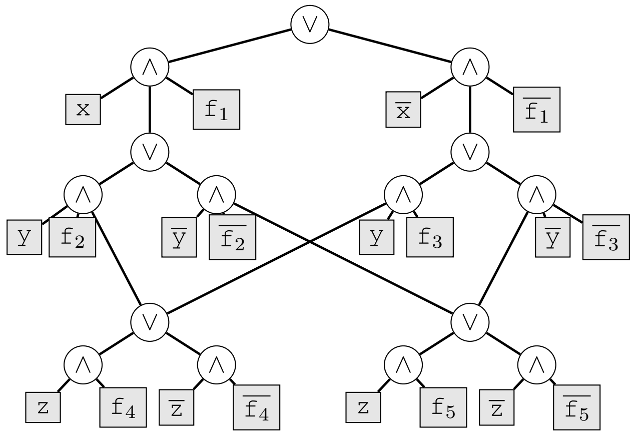
The set of observations compile to the conjunction of the compiled formulae for each expression observed. Programs compile to a list of formulae with an entry for each query. If the query is for the marginal probability of an expression, then the formula is the conjunction of the compiled statements, compiled observations, and the queried expression; if the query is for the marginal-MAP over some number of variables in the program, then the formula is just the conjunction of the statements and observations. Note that we can reuse the compiled statements and observations for every query, we will empirically show the benefits of this amortization in Figure 8.

3.4 Efficient Computation of `map` Upper-Bounds

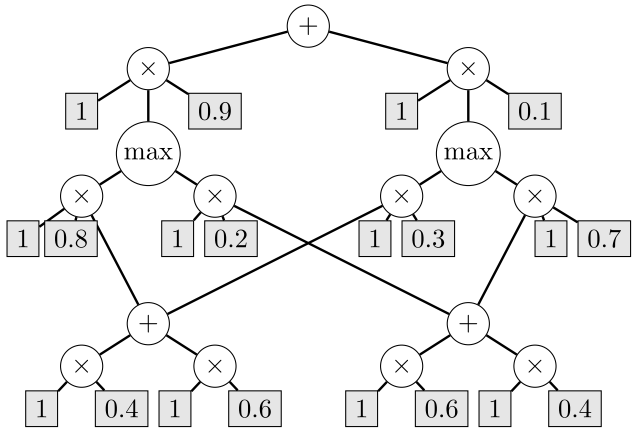
The previous section described how to associate `PineaPPL` programs with equivalent weighted Boolean formulae, but this in itself is not a helpful reduction: we still need to leverage this translation to perform efficient marginalization and `map` upper-bound computation. To do this, we compile the logical formula into a *probabilistic circuit* that enables efficient weighted model counting computation (Chavira and Darwiche 2008). There are a variety of ways of accomplishing this compilation step and a variety of compilation targets such as binary decision diagrams (BDDs) (Brace, Rudell, and Bryant 1990), SDDs (Darwiche 2011), and d-DNNF (Chavira and Darwiche 2008). In this paper we compile to BDDs, but in principle any of these other compilation targets can be used instead.

Figure 6a shows a logical circuit that encodes the logical formula in Equation 5. The circuit is read top-down. Each leaf of the circuit is a literal, and the internal nodes are connectives denoting logical disjunction and conjunction. The internal nodes of this circuit have particular properties that enable fast `WMC`. First, every disjunction is called a *decision*: each branch fixes variables to particular values, and hence every branch is logically mutually exclusive with every other branch. For instance, the left branch of the root node decides `x` and $\mathtt{f}_1$ are both `T`. Each conjunction is *decomposable*, meaning that any two branches of a conjunction refer to distinct sets of variables. Together, the decision property and decomposability enable efficient `WMC` computation in a single bottom-up pass over the circuit. See Darwiche and Marquis (2002) for further details on circuit properties.

Compiling an arbitrary `PineaPPL` program to a structured logical circuit that supports linear-time `WMC` is #P-hard in the size of the program (Roth 1996). Hence, this compilation step can be expensive, and the resulting circuit can be exponentially large in the program size. However, this compilation step is nonetheless extremely profitable, since (1) it is possible to compile interesting `PineaPPL` encodings; and (2) the `map` search procedure in Algorithm 1 amortizes the expensive compilation by performing many (linear-time) marginalization and upper-bound queries.



(a) `PineaPPL` logical circuit compilation.



(b) Circuit that computes an upper-bound on the MAP.

Figure 6: Compilations of the program in Figure 5.

Efficient marginal computation The compiled logical circuit can be used to compute any arbitrary marginal of the circuit in linear time in a manner similar to that described by Chavira and Darwiche (2008). In particular, we will use the circuit structure and an appropriate instantiation of each variable to describe the necessary sequence of sums and products for computing this marginal. As an example, consider computing the marginal probability that `x=T` using the circuit in Figure 6a. First, we replace every leaf with a numerical value: (1) replace each parameter variable $\mathtt{f}_i$ by its appropriate weight in the program; (2) replace every literal except `x` with the constant 1; (3) replace `x` with 1 and $\overline{\mathtt{x}}$ with 0. Then, each $\vee$ node performs a sum over its children, and each $\wedge$ node performs a product. Any (joint) marginal can be computed in linear time via an appropriate leaf substitution in this manner.

Efficient `map` upper-bound computation Now we wish to use this compiled circuit to compute the upper bound of a `map` state. First, we observe that if the `map` variables occur first in the decision order (i.e., near the root of the circuit), then we can compute the exact `map` state by simply applying the bottom-up pass described above and replace the relevant sums with products. Bellodi et al. (2020) observed that, by suitably restricting the compilation order for `ProbLog` to place all `map` variables at the front of the decision order, one can use this principle to design a (non-first-class) `map` inference procedure on the order-constrained circuit. However, this order-constrained approach has two downsides. First, if one wishes to perform multiple `map` queries (as in our setting), then one would be forced to compile the program many times with different orders for each `map` query; we show in Section 4 that this has performance implications. Second, as observed by Park and Darwiche (2003), order constraints have negative implications on the initial compilation performance.

Hence, we adopt a similar approach to Huang et al. (2006): we compile a single circuit with a good ordering (chosen heuristically based on the program order, falling back on lexicographic order when program order is inconsistent across `if-else` branches) without regard for the eventual `map` variables: this optimizes for fast initial compilation. Then, rather than computing the exact `map` state on the circuit, we use the circuit to compute an efficient upper-bound by commuting the order of sums and maxes and applying Jensen's inequality, as in Huang et al. (2006). This is visualized in Figure 6b, which is computing an upper-bound on the probability of the `map` state for variables `x` and $\mathtt{f}_1$. In general, to compute an upper-bound on the `map` state for a set of variables, an $(\vee)$ node that decides any `map` variable is interpreted as a $\max$ node; otherwise, the $(\vee)$ node is interpreted as a $+$ node. Therefore, *the same compiled circuit can be used for solving many different* `map` *queries by interpreting it in different ways* – this is our key observation, which is what makes this approach to `map` uniquely well-suited for PPLs that support a first-class `map` keyword.

4 Empirical Evaluation & Case Studies

Our goal in this section is to empirically evaluate the scalability of the `map` procedure outlined in the previous section, and compare against existing `map` implementations for PPLs. We implemented `PineaPPL` as described by the operational semantics in Section 3.3 and instantiated Algorithm 1 with the upper-bound and marginal computation described in Section 3.4. We compiled `PineaPPL` to BDDs using a variable order described previously. To the best of our knowledge there is only one existing PPL that supports `map` inference: `ProbLog` (Bellodi et al. 2020).[1]

4.1 Comparison with **`ProbLog`**

First, we compared `PineaPPL` against `ProbLog`. Like `PineaPPL`, `ProbLog` compiles to a logical circuit in order to perform `map` inference. However, `ProbLog` has the following key differences: (1) `map` in `ProbLog` is not first-class and can only be performed once every program run; (2) `ProbLog` is a logic-based rather than imperative PPL with a very different compilation encoding and strategy; and (3) `ProbLog` reduces `map` inference to utility maximization using DT-ProbLog (Van den Broeck et al. 2010) rather than branch and bound search.

One challenge with evaluation is that there is no standard set of `map` benchmarks for comparing the performance across PPLs. We developed a new selection of benchmarks based on discrete Bayesian networks and compiled these networks into equivalent `PineaPPL` and `ProbLog` programs.[2] The "Cancer" and "Sachs" networks are small enough to run `map` queries over the entire powerset of possible variables. For the "Alarm" and "Insurance" networks, we selected 5 variables uniformly at random, and ran the powerset of possible queries over those 5 variables. Figure 7 gives a cactus plot showing the relative performance of these two `map` inference algorithms on four selected Bayesian networks. These Bayesian networks are very challenging: the "Alarm" network is the largest, and it has several thousand random variables. To our knowledge, these are by far the largest probabilistic programs that exact `map` inference has been performed on. On two of the examples (Insurance and Alarm), `ProbLog` failed to complete a single `map` query within the time limit.

As mentioned in Section 3.3, `PineaPPL` can perform multiple `map` queries on a single program, reusing the compiled model and compiled observations for each query. To give the fairest comparison with `ProbLog`, which does not have this capability, for the experiments depicted in Figure 7 we had `PineaPPL` compile a new program for every single query. To demonstrate the effect that this model amortization has on the performance of `PineaPPL`, we ran a subsequent experiment where we included every query we tested at the end of a single program. Figure 8 shows percent speedup of running the single the single amortized program compared to the time it takes to run each query individually for a sequence of programs.

5 Related Work

Meta-inference Many PPLs today contain some support for forms of *meta-inference*: the ability to evaluate a marginal query while running a program. Examples include `Church` (Goodman et al. 2008), Anglican (Tolpin, Van de Meent, and Wood 2015), `Gen` (Cusumano-Towner et al. 2019), meta-`ProbLog` (Mantadelis and Janssens 2011), `Venture` (Mansinghka, Selsam, and Perov 2014), and `Omega` (Tavares et al. 2019). It is possible to use meta-inference to solve `map` by querying for the sub-distribution

[1]*Notes on reproducibility:* All benchmarks were conducted on the same server with 512GB of RAM and two AMD EPYC 7543 CPUs. `PineaPPL` was entirely implemented in the Rust programming language. Each experiment was allocated a single core. We will release all source code necessary for fully reproducing our experimental results upon publication. The source code for all experiments is provided in the supplement.

[2]See the supplementary material for the source programs. Bayesian networks were selected from https://www.bnlearn.com/bnrepository/.

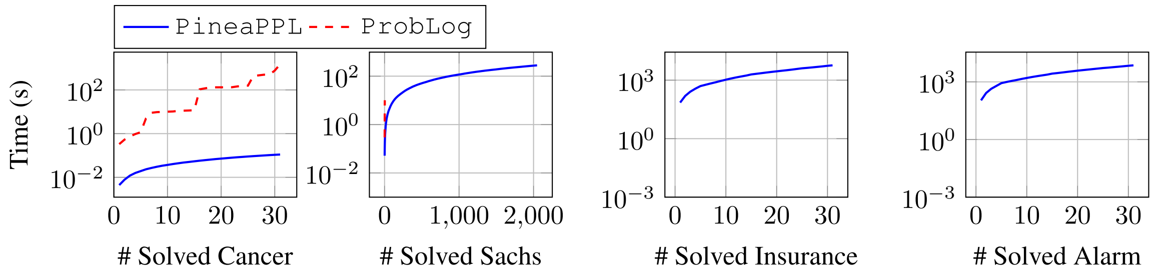


Figure 7: Cactus plots visualizing the number of solved benchmarks for `ProbLog` and `PineaPPL`. Plots without `ProbLog` results indicate that `ProbLog` failed to complete a single `map` query. Lower is better.

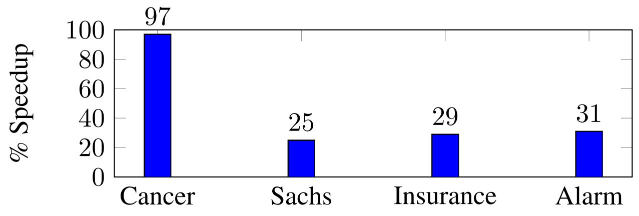


Figure 8: Speedups that result from amortization.

over all variables one wishes to marginalize over and enumerating that distribution to select the maximum value. However, this runs into a clear state-space explosion challenge: exhaustively enumerating the space of possible assignments during meta-inference is infeasible for many of the examples we showed in our experiments (for instance, the examples in our Bayesian network benchmarks query the `map` state of over 100 variables in some instances). Hence, for scalability reasons, we argue that a `map` query is an invaluable first-class citizen in addition to meta-inference.

Probabilistic soft logic (PSL) PSL is a probabilistic programming language purpose-built for computing the `map` state of a configuration of variables (Bach et al. 2017). PSL relaxes the probabilistic semantics in order to transform `map` estimation into a convex-optimization problem, allowing it to scale to extremely large problem instances. However, PSL does not support first-class `map` or observations, and furthermore does not have a precise probabilistic semantics, so it is not directly comparable to our approach in `PineaPPL`.

6 Conclusion

We introduced `PineaPPL`, an imperative PPL that compiles to probabilistic circuits and supports first-class `map` inference. By developing a suitable logical encoding for `PineaPPL`, we are able to leverage existing methods for exact `map` inference that were developed for Bayesian networks in order to do compositional `map` inference on probabilistic programs. We compared `PineaPPL` against existing `map` inference algorithms for `ProbLog` and showed `PineaPPL` can scale to orders-of-magnitude larger examples than are currently supported by the search-based `map` inference in `ProbLog`. For future work, we aim to (1) further specialize `map` by exploiting the repeated structure of similar `map` queries; (2) combine `map` inference with other more common forms of meta-reasoning like meta-inference; and (3) combine our exact discrete solver with continuous approximate methods to scale hybrid examples; and (4) explore circuit-transformation based approaches for possibly gaining more amortization benefits (Choi, Friedman, and Van den Broeck 2022).

References

Bach, S. H.; Broecheler, M.; Huang, B.; and Getoor, L. 2017. Hinge-loss markov random fields and probabilistic soft logic.

Bellodi, E.; Alberti, M.; Riguzzi, F.; and Zese, R. 2020. MAP inference for probabilistic logic programming. *Theory and Practice of Logic Programming*, 20(5): 641–655.

Borgström, J.; Gordon, A. D.; Greenberg, M.; Margetson, J.; and Gael, J. V. 2011. Measure transformer semantics for Bayesian machine learning. In *European symposium on programming*, 77–96. Springer.

Brace, K. S.; Rudell, R. L.; and Bryant, R. E. 1990. Efficient implementation of a BDD package. In *27th ACM/IEEE design automation conference*, 40–45. IEEE.

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

Choi, Y.; Friedman, T.; and Van den Broeck, G. 2022. Solving Marginal MAP Exactly by Probabilistic Circuit Transformations. In *International Conference on Artificial Intelligence and Statistics*, 10196–10208. PMLR.

Cusumano-Towner, M. F.; Saad, F. A.; Lew, A. K.; and Mansinghka, V. K. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, 221–236.

Darwiche, A. 2009. *Modeling and reasoning with Bayesian networks*. Cambridge university press.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

Darwiche, A.; and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17: 229–264.

De Campos, C. P. 2011. New complexity results for MAP in Bayesian networks. In *IJCAI*, volume 11, 2100–2106.

Fierens, D.; Van den Broeck, G.; Renkens, J.; Shterionov, D.; Gutmann, B.; Thon, I.; Janssens, G.; and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15(3): 358–401.

Goodman, N. D.; Mansinghka, V. K.; Roy, D.; Bonawitz, K.; and Tenenbaum, J. B. 2008. Church: A Language for Generative Models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, UAI'08, 220–229. Arlington, Virginia, USA: AUAI Press. ISBN 0974903949.

Holtzen, S.; Van den Broeck, G.; and Millstein, T. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. In *Proc. ACM Program. Lang.*, OOPSLA 2020, 140:1–140:31. Association for Computing Machinery.

Huang, J.; Chavira, M.; Darwiche, A.; et al. 2006. Solving MAP Exactly by Searching on Compiled Arithmetic Circuits. In *AAAI*, volume 6, 3–7.

Mansinghka, V.; Selsam, D.; and Perov, Y. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*.

Mantadelis, T.; and Janssens, G. 2011. Nesting probabilistic inference. *arXiv preprint arXiv:1112.3785*.

Mei, J.; Jiang, Y.; and Tu, K. 2018. Maximum a posteriori inference in sum-product networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Park, J. D.; and Darwiche, A. 2003. Solving MAP exactly using systematic search. *UAI*.

Rainforth, T.; Cornish, R.; Yang, H.; Warrington, A.; and Wood, F. 2018. On nesting monte carlo estimators. In *International Conference on Machine Learning*, 4267–4276. PMLR.

Roth, D. 1996. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2): 273–302.

Saad, F. A.; Rinard, M. C.; and Mansinghka, V. K. 2021. SPPL: probabilistic programming with fast exact symbolic inference. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 804–819.

Sang, T.; Beame, P.; and Kautz, H. A. 2005. Performing Bayesian inference by weighted model counting. In *AAAI*, volume 5, 475–481.

Stuhlmüller, A.; and Goodman, N. D. 2014. Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs. *Cognitive Systems Research*, 28: 80–99.

Tavares, Z.; Zhang, X.; Minaysan, E.; Burroni, J.; Ranganath, R.; and Lezama, A. S. 2019. The Random Conditional Distribution for Higher-Order Probabilistic Inference. *arXiv preprint arXiv:1903.10556*.

Tolpin, D.; Van de Meent, J.-W.; and Wood, F. 2015. Probabilistic programming in Anglican. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 308–311. Springer.

Tolpin, D.; and Wood, F. 2015. Maximum a posteriori estimation by search in probabilistic programs. In *International Symposium on Combinatorial Search*, volume 6.

Tolpin, D.; Zhou, Y.; Rainforth, T.; and Yang, H. 2021. Probabilistic programs with stochastic conditioning. In *International Conference on Machine Learning*, 10312–10323. PMLR.

Van den Broeck, G.; Thon, I.; Van Otterlo, M.; and De Raedt, L. 2010. DTProbLog: A decision-theoretic probabilistic Prolog. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, 1217–1222.