Scaling Integer Arithmetic in Probabilistic Programs

Abstract

Probabilistic programming languages (PPLs) allow a program to define a probabilistic model. Within the framework of a programming language, they provide the ability to declare random variables and condition on complex events. Much like traditional programming languages, where integers are perhaps the most common data type, one natural feature of PPLs is their ability to manipulate distributions over integers. Although most PPLs provide the ability to express distributions over integers, their support for inference over these distributions is limited. Most approaches essentially reduce to enumeration of all possible values, a strategy that does not scale well, or to general-purpose sampling. Our insight is that there is structure in arithmetic that these approaches are not using. We present a binary encoding strategy for random integers where binary-number bits are replaced by Boolean functions of random coin flips. This binary encoding then enables the circuit compilation approach to probabilistic inference to naturally exploit the structure behind arithmetic operations. This inference strategy scales to much larger integer distributions with arithmetic, further allowing us to support more language constructs such as parameterised flips and Beta priors, which enables Bayesian parameter learning from incomplete data.

1 INTRODUCTION

Probabilistic programming languages (PPLs) are expressive languages for defining probability distributions. The core idea of a PPL is to enrich a programming language with the ability to sample from and observe facts about probability distributions: hence, the program itself defines a probabilistic model. In this paper we focus on a particular programming feature: reasoning about random integers and integer arithmetic. Of course, integers are ubiquitous in traditional programs, and correspondingly it is very natural for probabilistic programs to manipulate distributions over integers, for example to model rankings, text manipulations, and distributions over graphs.

Most PPLs have the means to express distributions over integers as well as arithmetic operations, but unfortunately their support for reasoning about such programs is quite limited. Integer distributions are inherently *discrete*, so languages whose inference approaches rely on forms of differentiability typically resort to enumerating all possible discrete values [Gelman et al., 2015, Bingham et al., 2019, Dillon et al., 2017]. Further, even languages that support exact reasoning about discrete distributions typically model integer distributions as categoricals and integer operations as explicit transformations on such categoricals [Goodman and Stuhlmüller, 2014, Holtzen et al., 2020], which also reduces to value enumeration. As a result, existing PPLs do not scale well on programs that manipulate integers, thereby preventing their usage on many natural problems.

A commonality among these prior approaches is that they treat integer operations as *black boxes*. Our key insight is that integer arithmetic is itself inherently full of structure that can be exploited to scale probabilistic inference. In particular, the binary representation of integers and integer operations exposes this structure; by extending this binary encoding to manipulate *distributions* over integers, we naturally expose conditional independences that can be used to factorize inference.

In this paper, we develop a new exact inference strategy for integers based on this insight. We describe binary encodings of integer distributions, including a generic encoding as well as one that exploits the special structure in the uniform distribution on integer ranges. A core element of discrete probabilistic programs is the ability to represent a Bernoulli random variable, something which is often given the syntax `flip(p)`. While integers are traditionally represented as

Submitted to the 39th Conference on Uncertainty in Artificial Intelligence (UAI 2023). **To be used for reviewing only**.

```
id = [discrete([0.72, 0.01, 0.01, 0.01, 0.01,
                0.01, 0.2, 0.01, 0.01, 0.01]),
      ...,
      discrete([0.01, 0.01, 0.05, 0.01, 0.01,
                0.63, 0.2, 0.01, 0.01, 0.05])]
check_digit = id[0]
remaining_id = id[1:]
check_val = luhn_checksum(remaining_id)
observe((check_digit + (check_val % 10)) == 10)
return id
```

Figure 1: A probabilistic program for the student ID probabilistic inference problem using integer random variables (discrete), integer arithmetic (the Luhn algorithm function), and Bayesian conditioning (observe)

```
def luhn_checksum(id)
  sum = 0
  for i in 0..length(id)-1
    digit = id[i]
    if i % 2 == length(id) % 2:
      if digit > 4:
        sum += 2*digit - 9
      else:
        sum += 2*digit
    else:
        sum += digit
  return sum
```

Figure 2: Luhn algorithm implementation

sequences of bits, one can analogously replace these bits with logical expressions over `flips` to instead represent distributions over integers. Arithmetic can then be implemented within the program using standard logic circuits as are used in computer hardware.

Finally, we observe that exact inference methods for discrete probabilistic programs based on *knowledge compilation* [Holtzen et al., 2020, Raedt et al., 2007, Chavira et al., 2006, Chavira and Darwiche, 2008, Fierens et al., 2015b] naturally identify and exploit the structure exposed by this new probabilistic integer representation. The knowledge compilation approach reduces inference to weighted model counting (WMC) on weighted Boolean formulas (WBFs). Typically these formulas are represented using data structures such as binary decision diagrams (BDDs), which exploit conditional independences to represent distributions compactly and support linear-time WMC in the size of the BDD.

To evaluate our approach, we have implemented Dice [Holtzen et al., 2020] and its semantics as a shallow embedding in Julia [Bezanson et al., 2017] along with the support for binary encoding for integers. We refer to this implementation as `Dice.jl` in rest of the paper. We compare the performance of `Dice.jl` to that of other PPLs that support exact discrete inference, on a range of probabilistic models over integers taken from the literature and derived from natural examples. Our approach is the fastest in the majority of benchmarks and in many cases provides orders-of-magnitude speedups. We also empirically show that the gains from our approach of binarising integers generalize to other systems using knowledge compilation.

Finally, we show that scalable support for probabilistic models involving integers allows us to push the boundaries of discrete probabilistic programming systems in surprising ways. We demonstrate how to model a Beta distribution, a continuous distribution, using probabilistic integers. This modelling method exploits the conjugacy property of the Beta distribution, through which we can always characterize the distribution through its (integral) sufficient statistics. We demonstrate how this approach allows us to use the Beta distribution as a prior distribution, and apply it to the problem of Bayesian learning of Bayesian network parameters given incomplete data. We also explore how this model belongs to a general class of urn models [Mahmoud, 2008], which can also be used to represent other common distributions such as the negative binomial and hypergeometric.

The structure of this paper is as follows: Section 2 gives a motivating example for integer arithmetic. Section 3 explains our method of integer representation and explores how common integer operations on this representation have structure exploitable by knowledge compilation. Section 4 empirically evaluates our representation and compilation strategy against existing PPLs. Section 5 explores the representation of a Beta-Bernoulli process with random integers and demonstrates its application to the task of Bayesian network parameter learning. Sections 6 and 7 discuss related work and conclude respectively.

2 MOTIVATION

As a motivating example, consider the following probabilistic model based on student ID numbers. Suppose that an OCR system is attempting to parse a handwritten student ID number; for each digit of the ID, it gives us a probability distribution representing its beliefs about what the digit could be. Combining this output, we get a probability distribution over all possible student IDs.

The Luhn algorithm [dcode, 2023] is a commonly used method of validating various kinds of ID numbers including student IDs. Given a starting ID such as 70733428, the algorithm provides for a way to compute a sum over the ID, giving us a check digit (4) which is then appended to the original ID to get a final ID: 470733428. This ID is the one actually issued to a student; when provided with an ID, we can validate it by recomputing the sum and looking at the check digit.

We wish to use the fact that the student ID can be validated

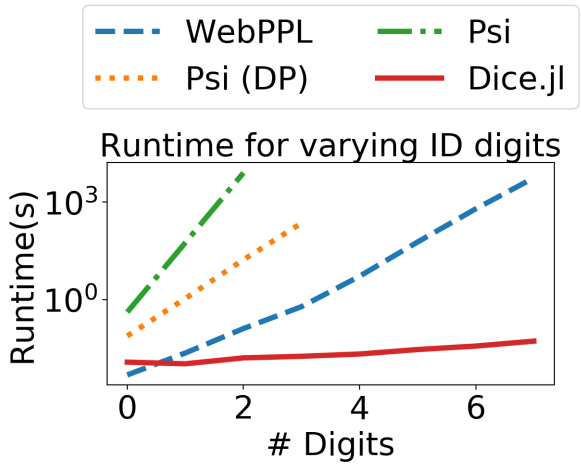

Figure 3: Single-marginal performance for ID example on increasing ID lengths. WebPPL and Psi scale exponentially due to having to enumerate all paths.

to additionally inform our single-digit distributions from the OCR system. We can implement this as a probabilistic program like the one in Figure 2. Figure 2 implements a function `luhn_checksum` that takes as input a list representing the digits of the student ID, excluding the check digit. It then does computation according to the Luhn algorithm to compute a sum over the digits, which is then returned. Figure 1 then uses this function: we create a list `id` which contains distributions over integers derived from the OCR system. The syntax `discrete(v)` for a vector $v = [p_0, .., p_n]$ creates a distribution over the numbers $0, .., n$, in which the number i has the probability p_i, and is used in our program to represent said OCR distributions. We can call the function and condition using the `observe` keyword in line 9 to get our updated distribution over IDs.

If we implement this program in today's probabilistic programming languages, we will run into a problem. Even if we only wish to compute the marginal probability over a single digit of the ID, Figure 3 shows that the runtime will scale exponentially in the number of digits in the student ID. Each additional digit will contribute a multiplicative amount to the number of total possible ID instantiations, meaning that any approach involving enumeration is inherently exponential. In practice, this means that the programs containing a realistic amount (9-10) of student ID digits will not be able to run.

The reality that straightforward programs like the one we just described fail to scale on existing probabilistic programming systems is the primary motivating factor behind our work. Figure 3 shows how our approach, implemented in the language `Dice.jl`, allows inference for such programs to scale for a larger, more realistic number of digits.

3 THE STRUCTURE OF INTEGER DISTRIBUTIONS

This section describes our key technical contributions, which exploit the inductive binary structure of integers to scale probabilistic inference. After some preliminaries, we provide a binary encoding of distributions on integers and highlight the resulting compact representation. We then show how the binary encoding can be leveraged to identify and exploit conditional independencies for inference over distributions that employ integer arithmetic.

3.1 PRELIMINARIES

Our work is built upon the knowledge compilation approach to discrete probabilistic inference [Holtzen et al., 2020]. This approach reduces inference on a discrete probabilistic program to weighted model counting (WMC) — a task commonly done on binary decision diagrams (BDDs), which support WMC in linear time on the BDD size. Therefore, the ability to scale inference is directly related to the ability to compile to a compact BDD. The BDD construction process identifies and exploits a ubiquitous property of distributions — conditional independencies — to reduce BDD size. In this section, we show that such independencies are widespread in integer distributions and thus can be exploited for scalable inference.

To that end, we employ a binary encoding of integer distributions, whereby such a distribution is represented as a tuple of Boolean random variables. Our approach will compile such tuples into *multi-rooted BDDs*, with one root per Boolean random variable in the tuple. Finally, we will encode integer operations as logical circuits that use this representation for its inputs and outputs.

3.2 ARBITRARY INTEGER DISTRIBUTIONS

To provide intuition for our encoding of integer distributions, consider the probability distribution π over the integers $\{0, 1, 2, 3\}$ such that $\pi(0) = 0.1, \pi(1) = 0.2, \pi(2) = 0.3$ and $\pi(3) = 0.4$. These probabilities can also be represented as a probability vector $[\pi(0), \pi(1), \pi(2), \pi(3)]$. We wish to represent this distribution using a tuple of two Boolean random variables (b_1, b_0) such that $\Pr(b_1 = 1, b_0 = 1)$ represents the probability of (1, 1) (the binary representation of 3), namely 0.4, and similarly for the other integers. That can be achieved by defining $b_1 = \text{flip}(\frac{7}{10})$ and $b_0 =$ if b_1 then $\text{flip}(\frac{4}{7})$ else $\text{flip}(\frac{2}{3})$. For example, $\Pr(b_1 = 1, b_0 = 1)$ is $(\frac{7}{10})(\frac{4}{7}) = 0.4$.

Using the above example, we would like to draw attention to two main insights. First, the most significant bit is a single coin flip whose probability is equal to the ratio of the elements in the second half of the probability vector

to the total sum of the vector. Second, conditioning on the most significant bit renders us two different distributions on smaller supports. Combining these insights, we arrive at Algorithm 1 shown below, which converts an arbitrary integer distribution into a binary encoding. The algorithm first computes the distribution on the most significant bit that is required, and then conditions on it to recurse on the two halves of the distribution. The algorithm assumes that the given distribution has an upper bound that is a power of 2, but we can always pad distributions that do not satisfy this with trailing zero probabilities.

Algorithm 1: INT_DIST($v \in \mathcal{R}^{2^b}$)

Input: Vector v such that $v_i = \text{pr}(i)$
Output: Vector of Boolean random variables of length b
$p \leftarrow \sum_{i=2^{b-1}}^{2^b-1} v[i] / \sum_{i=0}^{i=2^b-1} v[i]$
$f \leftarrow \text{flip}(p)$
pvec ← **if** f **then**
  INT_DIST($v[2^{b-1} : 2^b - 1]$)
else
  INT_DIST($v[0 : 2^{b-1} - 1]$)
end
return f :: *pvec*

We prove that Algorithm 1 is correct in the supplementary material. We also provide an upper bound on the size of the BDDs compiled from the resulting tuples of Boolean random variables for the variable order described in the algorithm.

Proposition 1 *For a discrete distribution over the integers* $\{0, 1 \ldots, 2^b - 1\}$*, the resulting tuple of Boolean random variables of INT_DIST compiles to a BDD of size* $\Theta(2^b)$.

Notably, an alternate approach in the literature for encoding categorical distributions such as integers [Sang et al., 2005] leads to a BDD that is at least larger by a linear factor. We provide more details of these proofs in the appendix.

Proposition 2 *A discrete distribution over integers when represented using the approach of Sang et al. [2005] gets compiled to a BDD of size* $\Theta(b2^b)$.

3.3 UNIFORM INTEGERS

A common kind of integer distribution is the uniform distribution over a specific range. While the encoding above handles arbitrary distributions, the structure of uniform allows for a special encoding with fully independent flips. Since the probability of every integer is equal, we can encode a uniform distribution over the integers $\{0, 1, \ldots 2^b - 1\}$ using a tuple of b independent unbiased coin flips ($\text{flip}(0.5)$). As an example, consider the uniform distribution over the integers $\{0, 1 \ldots 15\}$. Clearly, each possible instantiation of $(flip(0.5), flip(0.5), flip(0.5), flip(0.5))$ is equally likely, and thus equivalently the probability of each integer in the range.

Algorithm 2: UNIFORM(n)

Input: n, for a uniform distribution over $0, \ldots, n-1$
Output: Vector of Boolean random variables of length $\lfloor \log_2(n) \rfloor$
$b \leftarrow \lfloor \log_2(n) \rfloor$
if *flip* $\frac{2^b}{n}$ **then**
  return $[\text{flip}\, 0.5] * b$
else
  return 1::UNIFORM($n - 2^b$)
end

The method described above works for uniform distributions whose range is a power of 2; for ranges that are not a power of 2, we use the fact that any natural number can be expressed uniquely as sum of distinct numbers that are each powers of 2. This enables a uniform distribution over any range to be represented as a mixture of multiple uniform distributions over smaller power-of-two ranges. We capture this insight in Algorithm 2, which gives a method for representing uniform distributions starting at 0. The correctness of this approach is established in the supplementary material. Given this, we can then easily achieve any uniform by adding an offset.

3.4 INTEGER ARITHMETIC

While the binary representation of discrete and uniform distribution over integers is interesting, it does not by itself necessarily give much advantage. For example, if adding two such distributions still requires an explicit enumeration of all possible sums, then we have not gained much over the existing approaches to inference. However, the binary encoding enables us to leverage the structure of integers to do much better than this for common operations. In the following section, we demonstrate how this occurs for integer comparisons and addition.

3.4.1 Integer Comparisons

The comparison operator on binary tuples can be implemented using logic circuits like those in computer hardware. Suppose we compute $a < b$ for two binary numbers $a = 001$ and $b = 100$. The circuit first compares the most significant bits of these numbers, which are 0 and 1 respectively - enough to know that $a < b$ is true. If the two numbers instead had the same MSB, we would need to start this comparison over on the remaining bits. This process of computing $a < b$ highlights its key conditional independencies.

First, given the most significant bits (MSBs) of the operands are different, the result of $a < b$ does not depend on the remaining bits. Second, given the MSBs of the operands are the same, the computation on the remaining bits does not depend on the value of the MSBs. This structure gets automatically exploited when we use this standard logic circuit to compare integer distributions, where the inputs are now weighted Boolean formulas represented as BDDs, rather than bits.

More concretely, consider the following probabilistic program which defines two random variables having uniform distribution over the integers $\{0, 1, \ldots, 7\}$ and then outputs the probability of one integer being less than the other.

```
a = uniformInt(0, 7)
b = uniformInt(0, 7)
result = (a < b)
return result
```

Enumerating all the values that a and b can take in the above program would lead to 64 combinations. In contrast, the BDD for the comparison operation has size linear in the number of bits, due to its ability to exploit conditional independences. We later present empirical results demonstrating that this leads directly to better scalability for discrete inference.

3.4.2 Integer Addition

Consider two binary numbers $a = 001$ and $b = 100$ that we wish to add. Computing $a + b$ involves computing 4 new bits, each a function of the bits of a and b. The least significant bit of $a + b$ is computed as the *xor* of LSBs of a, b and 0 (the initial carry bit). The carry, computed as *and* of LSBs of a and b, would be passed on to the next bit and the same process will be repeated for the remaining bits. The above described process shows that given the carry bit, each bit of the result is independent of the lesser significant bits of the operands. As we showed earlier for the comparison operation, encoding arithmetic on integer distributions as a logical circuit directly exploits such conditional independences to produce a compact BDD, which in turn leads to significant performance gains. Wegener [2004] show that for an optimal variable ordering, there is a linear bound on the BDD for addition.

In this section, we described the structure of two arithmetic operations, namely comparison and addition independently. When composing these operations together, the compilation of weighted Boolean formulas will naturally compose as described in previous work [Holtzen et al., 2020]. The sizes of the resulting BDDs depend highly on the variable ordering. But even when the variable ordering is not optimal, conditional independences can still be identified and exploited, as shown by the experiments in the next section.

4 EMPIRICAL EVALUATION

In this section, we empirically test our integer compilation strategy. While we have demonstrated that a binary encoding exposes simple function structure that knowledge compilation can exploit, it remains to be seen if this can improve on current knowledge compilation methods. In addition, we have yet to show how our approach compares to other inference methods on larger, more complex arithmetic models. Therefore, we seek to answer the following two questions.

1. Does our binary encoding provide a benefit to existing languages using knowledge compilation?
2. How does our integer compilation approach compare to other methods on larger, more complex models?

4.1 INTEGER ENCODING AND COMPILATION

In this section, we demonstrate the benefit a binary encoding brings to knowledge compilation. To evaluate this, we implemented a binary encoded arithmetic representation in the PPL `Dice` [Holtzen et al., 2020] and the probabilistic logic programming language `ProbLog` [Fierens et al., 2015a], both of which use knowledge compilation as their approach to inference. We then compare the performance of this binary encoding to the native integer representation in each language, in `Dice`'s case a one-hot encoding. We do this comparison on the simple arithmetic operations and comparisons $a + b$, $a < b$, and $a == b$. Here, the runtime of each language corresponds to the time needed to compile and run inference on each representation, in effect measuring how well knowledge compilation based inference can exploit the structure of each function. We vary the size of the distributions over a and b so that we can see the scaling behavior with each implementation.

For the language `Dice`, we compare two primary items: `Dice` using its canonical one-hot encoding ("one-hot") and `Dice` using a binary encoding and with the corresponding arithmetic operations implemented as logical circuits ("binary"). We also include as reference `Dice.jl`, our system using the same binary encoding and knowledge compilation strategy as `Dice` instead embedded in the language Julia, which will be used in the following section ("Dice.jl"). The results are given in Figure 4. We can clearly see that our binary encoding outperforms the one-hot encoding, suggesting that it more readily allows for structure to be found.

For `ProbLog`, we also compare three items. Like before, we compare the native ProbLog integer representation ("native"), and a version using a binary encoding and logical circuits for arithmetic ("binary"). In addition, we now implement a hybrid form that represents numbers using a binary encoding but converts them to native ProbLog integers for arithmetic ("hybrid"). The results are given in Figure 5. We can see that the binary representation is much better than the

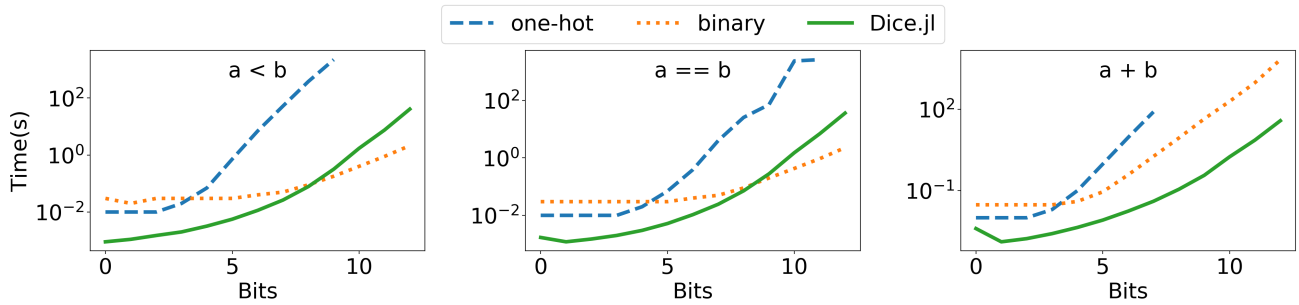

Figure 4: Dice performance on micro-benchmarks.

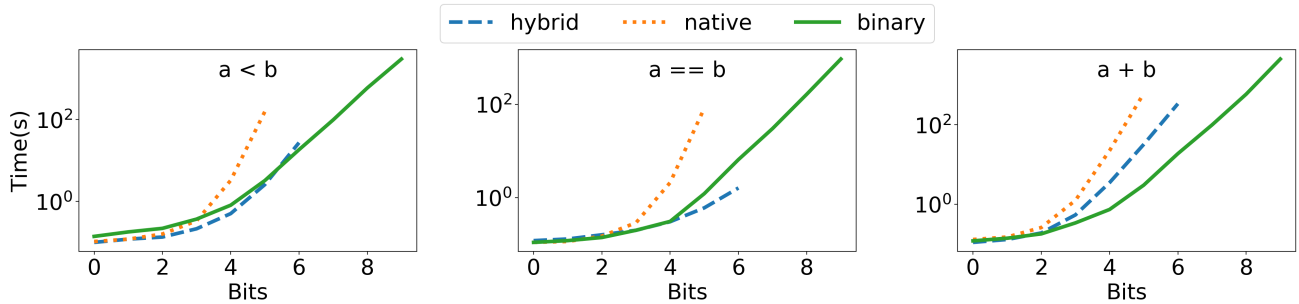

Figure 5: ProbLog performance on micro-benchmarks.

native implementation. Interestingly, the hybrid implementation is also better than the native implementation, and for the equality comparison better than even the circuit based approach; this suggests that even only using a binary representation (without special circuits for arithmetic operations) can give improvements with knowledge compilation.

It is thus clear that our binary encoding and circuit arithmetic, which are not used by existing PPLs using knowledge compilation, confer advantages in overall performance.

4.2 SCALING ON COMPLEX ARITHMETIC MODELS

To evaluate our approach on more complex models, we have implemented our integer representation in `Dice.jl`, a PPL embedded in Julia that is based off of `Dice` and uses the same knowledge compilation approach. In `Dice.jl`, unsigned random integers are implemented using the strategy described in the paper; signed random integers are additionally implemented as a natural extension. Provided are the syntax `discrete(..)` for arbitrary distributions and `uniform(..)` for uniform distributions, implemented using the algorithms in Section 3, as well as the operators for $=, <, +, -, *, /$, and $\%$. Simple operations are implemented as logical circuits, while more complex operations are implemented by composing simple ones.

We evaluate our approach on various models involving distributions over integers. The models evaluated on were taken from a variety of sources. These include examples involving integers from the existing PPL literature, various examples using continuous distributions adapted to discrete space, natural modelling tasks using integers such as ranking and text manipulation, and traditional algorithms in a probabilistic setting. A short description of our baselines and their sources are given in the supplementary material.

As a point of comparison, we use other two PPLs supporting exact discrete inference. We identify two major classes of exact inference approaches used for discrete probabilistic programs: enumerative methods, which work by enumerating all paths through the program, and symbolic methods, which represent and compute the probability distribution through symbolic expressions. We compare against `WebPPL` [Goodman and Stuhlmüller, 2014] from the former category and `Psi` Gehr et al. [2016] from the latter.

Our results are summarized in Table 1. Many of our benchmark models can naturally be scaled to different sizes; they are implemented in small, medium, and large (corresponding to model size) variants to display the scaling behavior. Psi supports two exact inference algorithms: a default symbolic exact inference algorithm ("Psi") and its specialized dynamic programming inference algorithm ("Psi (DP)"). Programs were evaluated to a 1 hour timeout.

We see that for the majority of baselines, our approach outperforms the current exact inference approaches, often achieving an orders-of-magnitude speedup. This validates the idea that our compilation strategy can outperform exist-

ing strategies by exploiting parts of the arithmetic structure.

We note that `Dice.jl` does not always outperform WebPPL, the enumeration based approach. We make special note of these examples. The caesar example introduces many random integers, but immediately makes an observation on their value, thereby reducing the enumeration task and making this tractable for all approaches. The disease example contain distributions on integers themselves parameterized by distributions on integers; for example, a binomial distribution with parameter n distributed by a uniform distribution. These distributions have much less structure to exploit, and so our approach becomes essentially enumerative, but with additional overhead in compilation. The GCD example, which makes repeated use of the mod ($\%$) operator, similarly has a harder to exploit structure. However, we can see in general that our approach scales well to larger examples, and outperforms the existing PPLs and inference approaches.

Table 1: Runtimes for probabilistic models using integers. x indicates a timeout (over 1 hour).

Benchmarks	Dice.jl	WebPPL	Psi (DP)	Psi
book	**5.297**	x	x	x
tugofwar	**0.106**	21.012	x	4599.845
caesar-small	**0.041**	0.074	2.022	402.196
caesar-medium	0.239	**0.135**	12.505	x
caesar-large	0.556	**0.227**	30.387	x
ranking-small	**0.007**	0.83	103.572	x
ranking-medium	**0.022**	x	318.658	x
ranking-large	**0.048**	x	330.51	x
radar1	**0.034**	118.002	394.525	2.517
floydwarshall-small	**0.009**	**0.009**	0.115	113.467
floydwarshall-medium	**0.515**	9.51	2792.14	x
floydwarshall-large	**3.406**	x	x	x
linear extensions-small	**0.003**	0.016	0.351	5.153
linear extensions-medium	**0.007**	0.465	111.38	x
linear extensions-large	**0.072**	162.009	x	x
triangle-small	**0.086**	3.693	616.746	482.14
triangle-medium	**0.455**	28.354	x	x
triangle-large	**17.365**	3863.421	x	x
gcd-small	2.876	**0.189**	24.33	x
gcd-medium	103.614	**2.501**	467.581	x
gcd-large	3745.9	**46.626**	x	x
disease-small	7.91	**1.093**	109.242	1009.848
disease-medium	764.212	**327.545**	x	x
luhn-small	**0.039**	0.428	44.164	x
luhn-medium	**4.575**	42.372	x	x

5 LANGUAGE CONCEPTS ENABLED BY INTEGER ARITHMETIC

Previous sections presented an inference strategy for integer distributions allowing for the scaling of integer arithmetic. In this section, we present a series of transformations in `Dice.jl` that demonstrate how these integers can be used to increase the expressiveness of the language.

5.1 FLIPS WITH RATIONAL PARAMETERS

Consider a parametrised Boolean random variable `flip(k/n)`, a flip with rational number bias where k and n are program variables. `Dice.jl` does not support this construct: the language requires the arguments to a flip to be a fixed constant at compile time. Since integers are the only numerical construct we have access to, we cannot represent such a division within the program.

To solve this, we make the key observation that $\text{uniform}(0, n) < k$ (where $\text{uniform}(0, n)$ is a uniform distribution over the integers $0, .., n-1$) has the same probability of being true as $\text{flip}(\frac{k}{n})$. We can therefore use this transformation to represent the parametrised flip. We recall that k can be a random number, and as long as n is deterministic, the uniform distribution and the less-than operator have compact representations, making this an efficient representation. A varying k means that we have a distribution over possible parameters, allowing us to now encode prior distributions over flip parameters.

5.2 BETA PRIORS AND BAYESIAN LEARNING

One natural prior for a Bernoulli random variable is the Beta distribution, its conjugate prior. How can we represent the continuous Beta distribution in discrete space? Suppose that we have a Beta distribution $\text{Beta}(a, b)$. A flip with parameter taken from this distribution will be true with probability the expectation of the distribution - $\frac{a}{a+b}$. We can therefore use our transformation from above to represent such a flip as $\text{uniform}(0, A+B) < A$, representing the parameters a and b as program variables A and B. As they are deterministic values, this is efficient.

This single flip by itself is not interesting - a prior distribution is meant to be used for learning. By the conjugate prior property of the Beta distribution, we know if this flip is true, we should have a posterior distribution of $\text{Beta}(a+1, b)$; if false, $\text{Beta}(a, b+1)$. We can represent this fact by including the parameters of the Beta distribution as program variables; we can then conditionally update these parameters based on the flip, incrementing A by 1 if true and B by 1 if false. Now, observing a value of this flip will give us the correct Bayesian posterior parameters for the Beta distribution in the variables A and B.

However, suppose we want to make multiple observations - we will need another flip! However, this conditional update means that A and B are no longer deterministic, meaning that our flip transformation will no longer be efficient. To address this, we make the observation that while our numerator A could take on the values a or $a+1$, depending on the outcome of the flip, our denominator A+B is either $(a+1)+b$, or $a+(b+1)$ - the same value! Therefore, we can keep track of our total denominator in an entirely

deterministic variable.

By repeating the flip and update steps given above, what we have actually implemented is a Beta-Bernoulli process via a Polya urn model - something analyzed in detail in probabilistic programs by Staton et al. [2018]. The flips we draw are equivalent to Bernoulli random variables with the same Beta-distributed parameter. This means that making an observation on the flip will implicitly update the underlying Beta prior. As an unobserved flip is true with probability equal to the expectation of the underlying Beta, we can get the expectation of the posterior distribution by computing the probability of this flip — a point estimate of the true Bayesian posterior.

5.3 BAYESIAN NETWORK PARAMETER LEARNING

One natural application of a Beta prior is for the parameters of a (binary) Bayesian network. The task of Bayesian network parameter learning can be described as follows: given a set of data consisting of instantiations of network variables, we want to find the network parameters maximizing the probability of this data. One interesting case is when our data is incomplete; that is, the instantiations have missing values. In this setting, a Bayesian approach to parameter learning must consider all (exponentially many) possible missing values.

This application naturally fits our probabilistic programming language with the process described above: we can easily model a Bayesian network with random flips drawn from Beta priors, representing our prior belief for each parameter. Observations on the network variables will then update our beliefs, from which we can get an expectation in the manner described above. We have implemented such an application on the `survey` [BNLearn, 2007] Bayesian network, on which we can scale to roughly 7 missing data points.

In the previous section, we described how a series of integer transformations allowed us to represent a Beta prior in a manner suitable for a learning task, an expansion of the expressive capability of `Dice.jl`. We note that this representation strategy — an implementation of an urn model — can also be used for many other distributions [Mahmoud, 2008], further increasing the space of distributions `Dice.jl` can handle.

6 RELATED WORK

A significant body of work is behind the development of probabilistic programming to its current state. Many existing probabilistic programming systems provide support for integers, but at a high level, our approach is the first one to handle probabilistic integers using their binary encoding. We survey the existing PPLs under the following categories.

Enumerative Exact Methods Numerous PPLs that support exact discrete inference over programs with integer distributions conduct inference by enumerating all the values of the distribution. Some notable examples include Goodman and Stuhlmüller [2014], Bingham et al. [2019]. Our approach does not involve enumeration but instead exploiting the binary structure of the integers.

Symbolic Exact Methods. There are algebraic methods for probabilistic inference that support integers by maintaining a closed form expression for the distribution. In notable systems like Psi [Gehr et al., 2016], these symbolic representations are essentially a mixture of delta distributions which still end up enumerating all the values of the distributions. There has been probability generating function based approaches as well which provide semantics for discrete programs with integers [Klinkenberg et al., 2023]. Our approach stands in contrast with these approaches as we do not resort to symbolic evaluation.

Approximate Inference Methods. There are multiple sampling based inference algorithms which work very well for probabilistic program with continuous distributions [Kantas et al., 2009, Hoffman and Gelman, 2014, Arouna, 2004]. In this work, we target the problem of exact inference for discrete probabilistic programs.

Graphical Model Based Inference Methods. There are existing works which compile probabilistic programs to probabilistic graphical models and further perform inference by factoring the distributions on PGMs [Pfeffer, 2009, McCallum et al., 2009]. These methods do support integers while treating them as categorical distributions. Our approach treats them as tuples of Boolean random variables with inductuve structure.

Knowledge Compilation Based Approaches. PPLs such as Holtzen et al. [2020], Raedt et al. [2007], Chavira et al. [2006], Chavira and Darwiche [2008], Fierens et al. [2015b] use a knowledge compilation approach to probabilistic inference, but none leverage our binary encoding for integer arithmetic.

7 CONCLUSION

We presented a strategy for encoding random integers in probabilistic programs involving a binary interpretation of random flips, where arithmetic operations can be done through standard Boolean circuits. When combined with the knowledge compilation approach to probabilistic inference, this strategy naturally exploits structure in arithmetic that current approaches do not account for. In practice, we showed that this allows for the computation of distributions on much more complex probabilistic models. One interesting consequence of this is that we can now represent the Beta distribution, a continuous distribution, in purely discrete programs.

References

Bouhari Arouna. Adaptative monte carlo method, a variance reduction technique. 10(1):1–24, 2004. doi: 10.1515/156939604323091180.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi: 10.1137/141000671.

Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

BNLearn. Bayesian networks, 2007. URL `http://bnlearn.com/bnrepository/discrete-small.html#survey`.

Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *J. Artificial Intelligence*, 172(6-7):772–799, April 2008. ISSN 0004-3702. doi: 10.1016/j.artint.2007.11.002.

Mark Chavira, Adnan Darwiche, and Manfred Jaeger. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning*, 42(1): 4–20, 2006.

dcode, 2023. URL `https://www.dcode.fr/luhn-algorithm`.

Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. Tensorflow distributions. *arXiv preprint arXiv:1711.10604*, 2017.

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. *Theory and Practice of Logic Programming*, 15: 358–401, 5 2015a. ISSN 1475-3081. doi: 10.1017/S1471068414000076.

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. *J. Theory and Practice of Logic Programming*, 15(3):358 – 401, 2015b. doi: 10.1017/S1471068414000076.

Timon Gehr, Sasa Misailovic, and Martin Vechev. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016.

Andrew Gelman, Daniel Lee, and Jiqiang Guo. Stan: A probabilistic programming language for bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, 40(5):530–543, 2015.

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2022-10-26.

Matthew D Hoffman and Andrew Gelman. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research*, 15 (1):1593–1623, 2014.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. In *Proc. ACM Program. Lang.*, OOPSLA 2020, pages 140:1–140:31. Association for Computing Machinery, 2020. doi: 10.1145/3428208.

N. Kantas, A. Doucet, S.S. Singh, and J.M. Maciejowski. An overview of sequential monte carlo methods for parameter estimation in general state-space models. *IFAC Proceedings Volumes*, 42(10):774–785, 2009. ISSN 1474-6670. doi: https://doi.org/10.3182/20090706-3-FR-2004.00129. 15th IFAC Symposium on System Identification.

Lutz Klinkenberg, Tobias Winkler, Mingshuai Chen, and Joost-Pieter Katoen. Exact probabilistic inference using generating functions. 2023.

Hosam Mahmoud. *Pólya urn models*. Chapman and Hall/CRC, 2008.

A McCallum, K Schultz, and S Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. *Proc. of NIPS*, 22:1249–1257, 2009. ISSN 03643417.

Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137, 2009.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog : A Probabilistic Prolog and Its Applications to Link. *Proc. of IJCAI*, pages 2468–2473, 2007.

Tian Sang, Paul Beame, and Henry Kautz. Performing bayesian inference by weighted model counting. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*, AAAI'05, page 475–481. AAAI Press, 2005. ISBN 157735236x.

Sam Staton, Dario Stein, Hongseok Yang, Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. The beta-bernoulli process and algebraic effects. 2018. doi: 10.4230/LIPICS.ICALP.2018.141.

Ingo Wegener. Bdds—design, analysis, complexity, and applications. *Discrete Applied Mathematics*, 138(1):229–251, 2004. ISSN 0166-218X. doi: https://doi.org/10.

1016/S0166-218X(03)00297-X. Optimal Discrete Structures and Algorithms.