Scaling Integer Arithmetic in Probabilistic Programs

William X. Cao[1] Poorva Garg*[1] Ryan Tjoa*[2] Steven Holtzen[3] Todd Millstein[1] Guy Van den Broeck[1]

[1]Department of Computer Science, University of California, Los Angeles, California, USA
[2] Department of Computer Science, University of Washington, Seattle, Washington, USA
[3]Khoury College of Computer Sciences, Northeastern University, Boston, Massachusetts, USA

Abstract

Distributions on integers are ubiquitous in probabilistic modeling but remain challenging for many of today's probabilistic programming languages (PPLs). The core challenge comes from discrete structure: many of today's PPL inference strategies rely on enumeration, sampling, or differentiation in order to scale, which fail for high-dimensional complex discrete distributions involving integers. Our insight is that there is structure in arithmetic that these approaches are not using. We present a binary encoding strategy for discrete distributions that exploits the rich logical structure of integer operations like summation and comparison. We leverage this structured encoding with knowledge compilation to perform exact probabilistic inference, and show that this approach scales to much larger integer distributions with arithmetic.

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 define, observe, and compute with random variables: hence, the program itself defines a probabilistic model. This paper focuses on a particular programming feature: scaling inference for programs with random integers and integer arithmetic. Integers are very challenging for today's approaches to probabilistic inference. The relationships between integer-valued random variables can be very complex: they can be added, multiplied, compared, etc. This rich structure is opaque to today's inference strategies. Trace-based sampling like Markov-Chain Monte Carlo, importance sampling, and sequential Monte-Carlo all collapse integer distributions to a single sampled point [Gelman et al., 2015, Bingham et al., 2019, Dillon et al., 2017, van de Meent et al., 2018, Lew et al., 2019]. These approximate inference strategies can scale well in many cases, but they struggle to find valid sampling regions in the presence of low-probability observations and non-differentiability (e.g., observing the sum of two large random integers to be a constant) [Gelman et al., 2015, Bingham et al., 2019, Dillon et al., 2017]. Exact inference strategies work by preserving the global structure of the distribution, but here there is a challenge: *what is the right strategy for efficiently representing and manipulating distributions on integers*? Today's PPLs that support exact inference and integer manipulation – such as Dice [Holtzen et al., 2020], ProbLog [De Raedt et al., 2007], Psi [Gehr et al., 2016], and WebPPL [Goodman and Stuhlmüller, 2014] – model integer distributions using what is essentially a one-hot categorical encoding (i.e., an integer distribution $[\texttt{0} \mapsto 0.25, \texttt{1} \mapsto 0.25, \texttt{2} \mapsto 0.25, \texttt{3} \mapsto 0.25]$ is represented simply as a vector). This encoding style is not capable of exploiting the structure of addition: adding two random variables effectively requires full enumeration.

Our first contribution is a new representation of distributions on integers as distributions on *binary encodings*. For instance, in the above example, rather than representing the distribution as an exhaustive map from integer values to probabilities, we represent it as a joint distribution on binary bits $[\texttt{00} \mapsto 0.25, \texttt{01} \mapsto 0.25, \texttt{10} \mapsto 0.25, \texttt{11} \mapsto 0.25]$. The upsides of this seemingly-equivalent representation are twofold. First, we can more efficiently represent the joint distribution itself when it has certain structure. In this case, because the distribution is uniform, we can represent it as a product of two independent Bernoulli distributions, one for each bit: we will show that the ability to factorize the distribution in this manner leads to significant performance improvements. Second, this binary representation reveals the structure of arithmetic: for instance, we can compare two integers by *independently* comparing each of their binary digits and aggregating the results.

Clearly a binary representation of integers reveals structure, but how can we automatically find and exploit this structure

*These authors contributed equally to this work.

Accepted for the 39th Conference on Uncertainty in Artificial Intelligence (UAI 2023).

```
1  id = [discrete([0.72, 0.01, 0.01, 0.01, 0.01,
2                  0.01, 0.2, 0.01, 0.01, 0.01]),...,
3        discrete([0.01, 0.01, 0.05, 0.01, 0.01,
4                  0.63, 0.2, 0.01, 0.01, 0.05])]
5  check_digit = id[0]
6  remaining_id = id[1:] //tail of the array
7  check_val = luhn_checksum(remaining_id)
8  observe((check_digit + (check_val % 10)) == 10)
9  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
    if i % 2 == length(id) % 2:
      if id[i] > 4:
        sum += 2 * id[i] - 9
      else:
        sum += 2 * id[i]
    else:
        sum += id[i]
  return sum
```

Figure 2: Luhn algorithm implementation

during inference in a PPL? As our second contribution, we show that two of today's PPLs – Dice [Holtzen et al., 2020] and ProbLog [De Raedt et al., 2007, Fierens et al., 2015] – are already capable of exploiting this structure if it is properly encoded into the program, by virtue of their *knowledge compilation* approach to inference. We give a lightweight strategy for encoding integer distributions, and show empirically that when using our new binary-encoded distributions these two languages scale to significantly larger and more complex integer distributions without essential modifications to their existing inference strategies.

As our third contribution we show that scalable support for random integer arithmetic 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. By doing so, we can use the Beta distribution as a prior for Bayesian learning.

The structure of this paper is as follows: Section 2 gives a motivating example for integer arithmetic. Section 3 explains our integer representation and explores how common integer operations on this representation have structure exploitable by knowledge compilation. Section 4 empirically evaluates our representation strategy against existing PPLs. Section 5 explores the representation of a continuous Beta prior with random integers. Sections 6 and 7 discuss related work and conclude respectively.

2 MOTIVATION

We begin with a motivating example highlighting how integer distributions are used in probabilistic programs. Consider the following probabilistic model based on student ID numbers. Suppose that an optical character recognition system is attempting to parse a handwritten student ID number. For each digit of the ID, it produces 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 [Luhn, 1960] is a commonly used method of validating various 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 prepended 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 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 call the `luhn_checksum` function on these integer distributions to get a distribution over checksums and condition using the `observe` keyword in line 9 to get an 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 programs containing student IDs of a realistic length (9-10 digits) will not run.

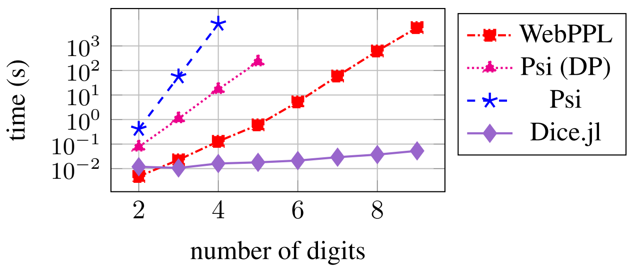


Figure 3: Single-marginal performance for ID example on increasing ID lengths. WebPPL and Psi scale exponentially due to having to enumerate all paths.

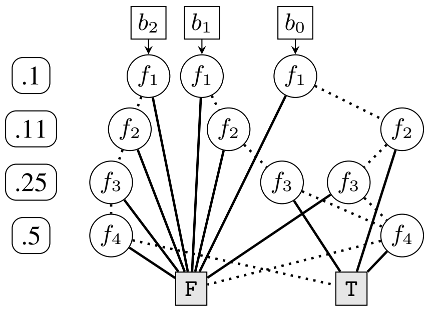
The fact that such straightforward programs fail to scale on existing probabilistic programming systems is the primary motivation behind our work. We have implemented our encoding of integer distributions in `Dice.jl`, a discrete PPL embedded in Julia that uses the same knowledge compilation approach as Dice [Holtzen et al., 2020]. Figure 3 shows that our technique allows inference for such programs to scale for a larger, more realistic number of digits.

3 REPRESENTING & MANIPULATING INTEGER DISTRIBUTIONS

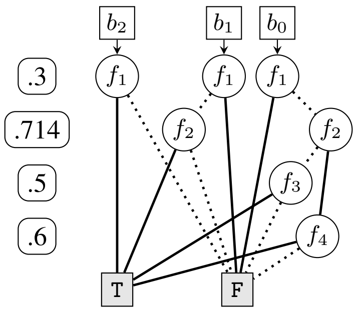
This section describes the key technical details behind a binary encoding approach and explains how such an encoding allows the knowledge compilation inference strategy used by Dice and ProbLog to automatically exploit arithmetic structure. We first provide a brief introduction to inference via knowledge compilation. We then demonstrate and analyze various approaches to constructing distributions over integers within probabilistic programs. Finally, we show how the binary encoding can be leveraged by knowledge compilation to identify and exploit contextual independencies for inference over distributions with integer arithmetic.

3.1 INTEGER DISTRIBUTIONS VIA BDDS

Thus far we have seen how binary-represented distributions expose structure and can enable effective scaling in practice. In this section we explain exactly how this performance improvement is achieved during inference. In particular, we show how *knowledge compilation* is capable of automatically finding and exploiting the structure of integer distributions and operations. Inference via knowledge compilation is currently the state-of-the-art approach to exact discrete probabilistic inference in certain classes of probabilistic programs [Holtzen et al., 2020, De Raedt et al., 2007, Chavira et al., 2006, Chavira and Darwiche, 2008, Fierens et al., 2015]. The heart of inference via knowledge compilation is a reduction from inference to *weighted model counting* (WMC). Let φ be a Boolean formula and w be a



(a) Multi-rooted BDD for a CATEG_INT encoded integer.



(b) Multi-rooted BDD for an BITWISE_INT encoded integer.

Figure 4: BDDs representing the integer distribution $[0 \mapsto 0.1, 1 \mapsto 0.1, 2 \mapsto 0.2, 3 \mapsto 0.3, 4 \mapsto 0.3]$ resulting from CATEG_INT and BITWISE_INT encoding methods. BITWISE_INT achieves a smaller BDD by compactly representing higher order bits.

map from literals in φ to real-valued weights; the pair (φ, w) is called a *weighted Boolean formula*. Then, the *weighted model count* $\texttt{WMC}(\varphi, w)$ is a weighted sum of models of φ:

$$\texttt{WMC}(\varphi, w) = \sum_{m \models \varphi} \prod_{\ell \in m} w(\ell). \qquad (1)$$

This reduction to WMC is not useful on its own however: the WMC task is #P-hard for an arbitrary Boolean formula φ. This is where knowledge compilation comes into the picture: φ is compiled into a data structure that supports efficient weighted model counting in the size of the data structure. A common example of such a knowledge-compilation data structure is a binary decision diagram (BDD), which supports linear-time WMC, but there are many others [Darwiche and Marquis, 2002]. PPLs like Dice and ProbLog work by compiling a program into a BDD or related compilation target and thereby reducing probabilistic program inference to WMC on that target [Chavira and Darwiche, 2005, Sang et al., 2005].

The cost of the knowledge compilation approach to inference is almost entirely determined by the structure of the program; the more structure that exists, the more compact the resulting BDD or related data structure can be. This leads to our core contribution: a new logical representation of integer distributions that is amenable to efficient compilation into BDDs. To demonstrate how BDDs can encode integer distributions, Figure 4 shows two different multi-rooted BDDs that represent *the same distribution* on integers. In both cases the roots in each BDD represent random variables for each binary digit: b_0 is the 0-order digit, b_1 is the 1-order digit. Weights of each positive literal are shown on the left (with the negative literal weight being 1 minus the positive literal weight); dotted edges represent a false assignment and solid edges represent true assignment. Intuitively, a binary representation has the potential to be more compact than a naive categorical representation due to the reduction in the number of roots: for instance, Dice [Holtzen et al., 2020] requires one root for each possible integer value.

As an example of how to use these data structures, consider computing the marginal probability of the high-order bit b_2 being true. This is $\texttt{WMC}(b_2, w)$, which is 0.3 – that is clear in Figure 4b since the sole path from b_2 to the true node has weight 0.3, but it is also true for the sole path from b_2 to the true node in Figure 4a, which has weight $(0.9 * 0.89 * 0.75 * 0.5) \approx 0.3$. In general, to compute the probability of an arbitrary integer, we convert it into binary and conjoin the appropriate roots: for instance, to compute the probability of the integer `0`, we compute $\texttt{WMC}(\overline{b_0} \wedge \overline{b_1} \wedge \overline{b_2}, w)$.

3.2 INTEGER ENCODINGS

The previous section demonstrated the potential for binary encodings in knowledge compilation, but how do we connect this to probabilistic programs? In this section we give lightweight encoding strategies for translating `discrete(..)` syntax for an arbitrary distribution over integers into distributions on Booleans, which knowledge-compilation-based languages like Dice and ProbLog are already capable of representing. How might such distributions be represented in a probabilistic programming language in practice? To make the problem concrete, we define the integer representation problem as follows: given an input vector $[p_0, .., p_w]$, we want a method which returns a distribution over integers taking on value i with probability p_i. While this is relatively restricted by demanding that our distribution is contiguous with lowest value 0, we can convert this to other distributions (for example) by adding an offset or multiplying by a constant.

3.2.1 A First Approach

One natural way of constructing such a categorical distribution, is as a set of if-else statements, with each branch corresponding to a different value. For example, the following probabilistic program snippet would correspond to the integer distribution with probability vector $[0.1, 0.2, 0.3, 0.4]$. The syntax flip(θ) used in the program is commonly used in discrete PPLs to represent a Bernoulli random variable with bias θ.

```
if flip(0.1) // Bernoulli(0.1)
   return 0
elseif flip(0.2/0.9)
   return 1
elseif flip(0.3/0.7)
   return 2
else
   return 3
```

We use a sequence of these random flips as arguments to the if-else statements to generate the mixture of numbers; note that we renormalize the flip probability at each step to get the correct distribution. This approach is generalized in Algorithm 1. This and future algorithms should be interpreted as a general method to represent a distribution over integers in any probabilistic programming language supporting Bernoulli random variables and (non-probabilistic) integers. Note that representing a categorical variable in this way is a probabilistic program framing of the SBK encoding presented by Sang et al. [2005].

Algorithm 1: CATEG_INT ($v \in [0,1]^w$)

```
Input: Vector $v$ such that $v[i] \propto$ pr($i$)
if $w$ == 1 or flip $\left(\frac{v[0]}{\sum v}\right)$ then
    return 0
else
    // Recurse on the remainder of $v$
    return 1 + CATEG_INT($v$[1:])
```

What would occur if we use Algorithm 1 to represent a distribution over binary-encoded integers in a language such as Dice? The BDD for one distribution represented using this approach is given in Figure 4a. Note that for each bit, the decision diagram is essentially a linear chain; intuitively, this corresponds to checking each if-else guard in sequence. There is almost no node reuse occurring in this BDD.

3.2.2 A More Compact Encoding

We propose an alternative method of representing integers from a probability vector that produces provably more compact BDDs. Rather than constructing our mixture by a linear pass through the probability vector, we can instead divide the vector into two parts, using a divide-and-conquer approach. Consider the same example as above, where we are again given as input a probability vector $[0.1, 0.2, 0.3, 0.4]$. To get the wanted distribution, we can conditionally add the value 2 with probability $\frac{0.3+0.4}{0.1+0.2+0.3+0.4}$, corresponding to the latter half of the vector. Depending on if 2 is added, we then conditionally add the value 1, with probability derived from the subvectors $[0.1, 0.2]$ and $[0.3, 0.4]$. An example program implementing this is given below:

```
num = 0
if flip(0.7) // 0.3 + 0.4
   num += 2
   if flip(0.4/0.7)
      num += 1
else
   if flip(0.2/0.3)
      num +=1
return num
```

This approach is formalized in Algorithm 2. For the sake of simplicity, we assume the input vector is always of length 2^b for some number b; this means that we always divide the vector into its two halves. For an arbitrary input vector, we can simply pad 0 probability values to fulfill this condition; in practice, this algorithm can easily be adapted to work without this explicit padding.

```
Algorithm 2: BITWISE_INT ($v \in [0,1]^{2^b}$)
Input: Vector $v$ such that $v[i] \propto$ pr($i$)
$p \leftarrow \frac{\sum_{i=2^{b-1}}^{2^b-1} v[i]}{\sum_{i=0}^{2^b-1} v[i]}$
if $length(v) == 1$ then
  return 0
else
  if flip($p$) then
    // Recurse on second half $\geq 2^{b-1}$
    return BITWISE_INT($v[2^{b-1} : 2^b]$) + $2^{b-1}$
  else
    // Recurse on first half $< 2^{b-1}$
    return BITWISE_INT($v[0 : 2^{b-1}]$)
```

Note that while Algorithm 2 uses arithmetic to produce the distribution, it only ever adds or return powers of two, which directly correspond to the bits of the integer. Therefore, when implementing the algorithm as a distribution on a tuple of bits, we encode each such addition by simply setting the appropriate bit. For the same example as above, our implementation constructs a tuple of bits (b_1, b_0) corresponding to a binary number such that $b_1 = \text{flip}(0.7)$ and $b_0 = \text{if } b_1 \text{ then flip}(\frac{0.4}{0.7}) \text{ else flip}(\frac{0.2}{0.3})$.

How does this method of representing integer distributions differ than the one given before? To see this, we look at the BDD for a distribution written in this manner given in Figure 4b. We can see a clear difference between this BDD and that for the approach given in Algorithm 1. The most significant bit corresponds to a BDD depending on only one flip, as this corresponds to the largest power of two: only one flip is used to determine its value. For the less significant bits, we add an additional layer of variables for each one, with the number of layers in total corresponding to the number of bits needed to represent the input distribution. This is in contrast to the CATEG_INT encoding, which requires checking a linear chain of variables for each bit, and so achieves a much more compact BDD representation.

We formalize this difference in BDD size in Proposition 1. Note that variable order can greatly influence the size of a BDD, and finding the optimal variable order is an NP-hard problem [Meinel and Theobald, 1998]; we follow the Dice convention of ordering logical variables using (strict, left-to-right) evaluation order. For example, in Figure 1, the Boolean variables encoding the `discrete` distribution on Line 1 occur before the variables in the distribution on Line 2 in the order.

Proposition 1. *A discrete distribution over the integers* $\{0, 1 \ldots, 2^b - 1\}$ *compiles to a BDD of size* $\Theta(b2^b)$ *when represented using CATEG_INT (Algorithm 1) and a BDD of size* $\Theta(2^b)$ *when represented using BITWISE_INT (Algorithm 2), with variables in flip evaluation order.*

It is BITWISE_INT that we have implemented in `Dice.jl` and experimentally evaluate in the next section.

3.2.3 Uniform Integers

The previous encoding strategy works for arbitrary distributions on integers, but in practice one often encounters common highly-structured distributions such as the uniform. One advantage of our approach is that we can exploit the structure of such distributions in order to scale significantly better than the general approach presented in Algorithm 2. In particular, the structure of the uniform distribution allows for a special encoding with fully independent flips.

Since the probability of every integer is equal, we can encode a uniform distribution over integers $\{0, 1, \ldots 2^b - 1\}$ by adding the values $2^0, 2^1, \ldots, 2^{b-1}$ independently with probability 0.5. From a bitwise perspective, this is same as independently setting each bit of the number to be true with probability 0.5. As an example, consider the uniform distribution over integers $\{0, 1 \ldots 15\}$. Clearly, each possible instantiation of $(\mathit{flip}(0.5), \mathit{flip}(0.5), \mathit{flip}(0.5), \mathit{flip}(0.5))$ is equally likely, and thus equivalently the probability of each integer in the range.

```
Algorithm 3: UNIFORM(n)
Input: Positive integer $n$
$b \leftarrow \lfloor \log_2(n) \rfloor$
if flip $\left(\frac{2^b}{n}\right)$ then
  $sum \leftarrow 0$
  for $i \leftarrow 0$ to $b-1$ do
    if flip $\left(\frac{1}{2}\right)$ then
      $sum \leftarrow sum + 2^i$
  return $sum$
else
  return UNIFORM($n - 2^b$) + $2^b$
```

The method described above works for uniform distributions whose range is 2^n for some n; for ranges that are not a power of 2, we use the fact we can decompose any natural number into a sum of 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 formalize this idea in Algorithm 3, which gives a method for representing uniform distributions starting at 0; the correctness of this approach is shown in the appendix. We can then use this approach to achieve any uniform distribution by adding an offset. Just like the previous algorithms, this algorithm is implemented by constructing sequences of bits in a manner equivalent to arithmetic.

We note that unlike the BITWISE_INT algorithm, where less significant bits have a dependence on more significant bits, our uniform algorithm leverages independence between the bits. Therefore, the BDD obtained when using UNIFORM is more compact than for our other algorithms, and fewer variables are needed to represent such a distribution.

3.3 EFFICIENT INTEGER OPERATIONS

While the binary representations of discrete and uniform distributions over integers are interesting, they do not by themselves necessarily give much advantage. If adding two such distributions still requires an explicit enumeration of all possible sums, then we have not gained much over the existing inference approaches. However, the binary encoding enables us to leverage the structure of integers to do much better than this for common operations. In this section, we demonstrate this for integer comparisons and addition.

3.3.1 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 (MSBs) 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 remaining bits. This process of computing $a < b$ highlights its key contextual independencies. First, given the 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 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 a uniform distribution over the integers $\{0, 1, \ldots, 7\}$ and then outputs the probability of one integer being less than the other.

```
a = uniform(0, 8)
b = uniform(0, 8)
return (a < b)
```

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, as it exploits contextual independences. We later present empirical results demonstrating that this leads directly to better scalability for discrete inference.

3.3.2 Integer Addition

Consider two binary numbers $a = 001$ and $b = 100$ that we wish to add. The least significant bit (LSB) of $a + b$ is computed as the *xor* of the LSBs of a, b and 0 (the initial carry bit). The carry, computed as the *and* of the LSBs of a and b, is passed on to the next bit and the same process will be repeated for the remaining bits. The process described above shows that given the carry bit, each bit of the result is independent of the lesser significant bits of the operands. Similar to the comparison operation, encoding addition on integer distributions as a logical circuit directly exploits such contextual independencies to produce a compact BDD, which in turn leads to significant performance gains. The manner in which addition corresponds to a compact BDD has been explored before; 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, 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, contextual independences can still be identified and exploited, as shown by the experiments in the next section.

4 EMPIRICAL EVALUATION

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

1) Does a binary encoding benefit existing knowledge compilation based languages?
2) Does our approach outperform those of existing PPLs that support exact discrete inference?

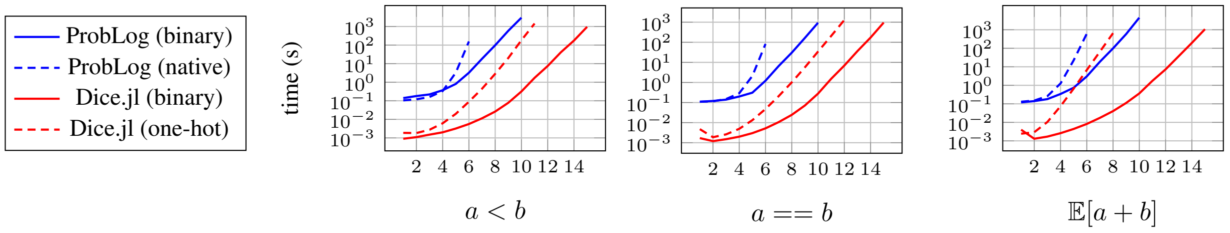


Figure 5: Time needed to compute the given operation on two random integers with varying bitwidth (x-axis).

To this end, we have implemented our integer representation in `Dice.jl`, a PPL embedded in Julia that uses the same knowledge compilation approach as Dice [Holtzen et al., 2020]. In `Dice.jl` (binary), unsigned random integers are implemented using the strategy described in the previous section; signed random integers are additionally implemented as a natural extension. The language provides the syntax `discrete(..)` for arbitrary integer distributions and `uniform(..)` for uniform distributions, implemented using the algorithms in Section 3, as well as the operators $=, <, +, -, *, /$, and $\%$. Simple operators are implemented as logical circuits, while more complex operators are implemented by composing simpler ones. [1]

Reported runtimes are a median over at least 5 runs; all experiments were run with a 1 hour timeout. A best-effort attempt was made for each language to implement benchmarks in a maximally performant manner. More experimental details are available in the appendix.

4.1 IMPROVING KNOWLEDGE COMPILATION LANGUAGES WITH A BINARY ENCODING

In this section, we demonstrate the benefit a binary encoding brings to knowledge compilation based languages. We compare our integer representation method to the methods of the PPL Dice [Holtzen et al., 2020] and the probabilistic logic programming language ProbLog [Fierens et al., 2015], both of which use knowledge compilation as their approach to inference. We do this by comparing the time needed to compute the simple arithmetic operations $a + b$, $a < b$, and $a == b$ on random integers of width 2^n for varying n from 1 to 15. Here, the runtime of each method corresponds to the time needed to compile and run inference on each representation, effectively measuring how well the knowledge compilation based inference can exploit the structure of each simple function. For addition, we use the expectation of the sum as our target computation to avoid an output distribution with an exponentially increasing support.

To allow for a fair comparison between ProbLog's native integer representation and our binary representation, we implement equivalent ProbLog programs computing the arithmetic operations, one using native ProbLog encodings and one using a binary representation. Both programs can then be run using ProbLog, controlling for the specific knowledge compilation system. To compare with Dice's native one-hot integer encoding, we implement both the one-hot encoding and our binary encoding in `Dice.jl`. We then run the simple arithmetic programs using both encodings.

The results of these experiments are presented in Figure 5. We can clearly see that our binary encoding outperforms the existing integer representation strategy used in each language; while at small distribution widths (on the order of 2^4), they are roughly comparable, our approach scales much better to larger integer distributions.

4.2 COMPLEX ARITHMETIC MODELS

We also evaluated `Dice.jl` on more complex models involving distributions over integers. The models were taken from a variety of sources. These include examples involving integers from the existing PPL literature, various examples using continuous distributions adapted to a 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 is given in the appendix.

As a point of comparison, we use two other 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.

We also compare against a version of `Dice.jl` that uses a one-hot encoding of integer distributions as a proxy for existing knowledge compilation approaches; this comparison avoids language-specific differences in performance. Compiled BDD sizes for the programs are provided in the appendix as an additional metric.

[1]Our implementation and code for all experiments are available at `https://github.com/Juice-jl/Dice.jl/tree/arithmetic`.

Table 1: Runtimes in seconds for probabilistic models using integers in various PPLs. ✗ indicates a timeout (over 1 hour).

Benchmarks	Dice.jl (binary)	Dice.jl (one-hot)	WebPPL	Psi (DP)	Psi
book	**5.297**	✗	✗	✗	✗
tugofwar	**0.106**	2660.373	21.012	✗	✗
caesar-small	**0.041**	4.968	0.074	2.022	402.196
caesar-medium	0.239	39.518	**0.135**	12.505	✗
caesar-large	0.556	122.109	**0.227**	30.387	✗
ranking-small	**0.007**	0.025	0.83	103.572	✗
ranking-medium	**0.022**	0.077	✗	318.658	✗
ranking-large	**0.048**	0.150	✗	330.51	✗
radar1	**0.034**	0.664	118.002	394.525	2.517
floydwarshall-small	**0.009**	0.152	**0.009**	0.115	113.467
floydwarshall-medium	**0.515**	624.220	9.51	2792.14	✗
floydwarshall-large	**3.406**	✗	✗	✗	✗
linear extensions-small	**0.003**	0.004	0.016	0.351	5.153
linear extensions-medium	**0.007**	0.013	0.465	111.38	✗
linear extensions-large	**0.072**	0.164	162.009	✗	✗
triangle-small	**0.086**	102.544	3.693	616.746	482.14
triangle-medium	**0.455**	1123.171	28.354	✗	✗
triangle-large	**17.365**	✗	✗	✗	✗
gcd-small	2.876	✗	**0.189**	24.33	✗
gcd-medium	103.614	✗	**2.501**	467.581	✗
gcd-large	✗	✗	**46.626**	✗	✗
disease-small	7.91	✗	**1.093**	109.242	1009.848
disease-medium	764.212	✗	**327.545**	✗	✗
luhn-small	**0.039**	0.594	0.428	44.164	✗
luhn-medium	**4.575**	23.933	42.372	✗	✗

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)").

For the majority of benchmarks, our approach outperforms the current exact inference approaches, often achieving an orders-of-magnitude speedup. This result empirically validates the ability of our compilation strategy to exploit arithmetic structure in order to improve inference performance. We observe that the binary encoding outperforms the one-hot encoding with the same underlying knowledge compilation approach, demonstrating its superiority in exposing 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 contains parametric 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 existing PPLs that support exact discrete inference.

5 ENABLING CONTINUOUS PRIORS WITH DISCRETE DISTRIBUTIONS

Previous sections presented an inference strategy for integer distributions allowing for the scaling of integer arithmetic. We now demonstrate an interesting application of these integers: representing the continuous Beta distribution in a discrete space. Continuous priors are an essential part of Bayesian reasoning. One particularly useful prior for discrete PPLs like `Dice.jl` is the *Beta prior* $\mathrm{Beta}(\alpha, \beta)$, which is conjugate to the Bernoulli. The Beta distribution is continuous and thus not amenable to direct representation in `Dice.jl`. However, we observe that if a Beta prior for a Bernoulli random variable has integral parameters α and β, then the posterior distribution is also a Beta with integral parameters. This section explains how we use this observation to represent Beta distribution in `Dice.jl`.

Assume we have the following program where `Dice.jl` is extended to permit restricted classes of Beta priors, where the parameters α and β must be constant integers:

```
θ = Beta(1, 2)
x = flip(θ)
observe(x)
return θ
```

We can perform inference for the posterior by exploiting well-known conjugacy results between Beta priors and Bernoullis. In particular, observing that x is true, as is done on Line 3 above, increases the *pseudocount* α by 1, making the posterior for θ become $\mathrm{Beta}(2, 2)$. Similarly, observing that x is false increases the pseudocount β by 1.

To automate this approach in `Dice.jl`, we introduce program variables A and B to represent the pseudocounts α and β, respectively. We then conditionally update these pseudocounts after each `flip`: increment α if the `flip` returns true; otherwise increment β. Doing so ensures that later observations will have the desired effect on the pseudocounts.

The only remaining challenge is that discrete PPLs that employ knowledge compilation, like `Dice.jl`, only support `flip`s whose parameters are constants, so the `flip`(θ) on Line 2 above is not supported. To encode it, we use the fact that $\mathbb{E}[\mathrm{Beta}(\alpha, \beta)] = \frac{\alpha}{\alpha+\beta}$, so the `flip` on line 2 can be simplified to `flip`$(\frac{A}{A+B})$. Unfortunately, `Dice.jl` still does not support this construct, since $\frac{A}{A+B}$ is not a constant. However, $A + B$ is always a deterministic integer, since each observation increases it by exactly 1. Therefore, we can introduce a variable T representing A+B and encode `flip`$(\frac{A}{A+B})$ as $\mathrm{uniform}(0, T) < A$ (where $\mathrm{uniform}(0, n)$ is the uniform distribution over the integers $0, .., n - 1$). Finally, we observe that it is not necessary to maintain both variables A and B, since B is derivable from A and T. The final transformed version of the program is as follows:

```
1  A = 1, T = 3
2  x = uniform(0, T) < A
3  A = if x then A+1 else A
4  T = T + 1
5  observe(x)
6  return (A, T-A)
```

If we wish to draw another flip on the same Beta prior, we can simply repeat the code in lines 2-4 above. In this way, 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]. We also note that this representation strategy — an implementation of an urn model — can also be used for many other distributions in addition to the Beta [Mahmoud, 2008]. An example application of this Beta prior — learning Bayesian network parameters — is given in the appendix.

6 RELATED WORK

PPL Inference. Knowledge-compilation-based PPLs are most closely-related to this work [Holtzen et al., 2020, De Raedt et al., 2007, Fierens et al., 2015, Saad et al., 2021, Pfanschilling et al., 2022]. All these languages stand to benefit from our new binary-encoding. Other PPLs perform exact discrete inference by eliminating discrete variables via enumeration or variable elimination; these approaches lose global structure and hence cannot exploit arithmetic structure as in our approach [Goodman and Stuhlmüller, 2014, Bingham et al., 2019]. Symbolic methods support integers by representing them as a symbolic formula or program [Gehr et al., 2016, Narayanan et al., 2016]; we believe that in principle it may be possible to adapt these symbolic representations to use a binary representation, but currently these systems do not. Recent work uses probability generating functions (PGFs) to represent (potentially unbounded) discrete distributions [Klinkenberg et al., 2023]. PGFs represent the distribution symbolically, but do not appear to be compatible with our strategy for binary encodings. Sampling based inference algorithms work very well for probabilistic programs with continuous distributions, but do not exploit the global structure of integer arithmetic [Kantas et al., 2009, Hoffman and Gelman, 2014, Arouna, 2004, Wu et al., 2016]. Finally, there are algorithms that seek to efficiently model integer distributions with recursion [Knuth and Yao, 1976, Saad et al., 2020]; these approaches are orthogonal to ours, as we do not use recursion.

Graphical models. Probabilistic graphical model (PGM) based inference methods [Pfeffer, 2009, McCallum et al., 2009, Sanner and Abbasnejad, 2012, Koller and Friedman, 2009] support integers by treating them as categorical distributions. PGMs struggle to represent arithmetic: for instance, a CPT for adding two n-bit numbers requires $O(2^n)$ entries.

7 CONCLUSION

We presented a strategy for encoding random integers in probabilistic programs via a binary representation, which allows arithmetic operations to be performed 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. We showed empirically that this allows existing discrete PPLs to scale to significantly more complex probabilistic models. One interesting consequence is that we can now leverage conjugacy to represent the Beta distribution, a continuous distribution, in purely discrete programs.

Acknowledgements

This work was funded in part by the DARPA PTG Program under contract number HR00112220005, NSF grants #IIS-1943641, #IIS-1956441, #CCF-1837129, #CCF-2220408, and a gift from RelationalAI. GVdB discloses a financial interest in RelationalAI.

References

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

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.

Mark Chavira and Adnan Darwiche. Compiling Bayesian networks with local structure. In *IJCAI*, pages 1306–1312, 2005.

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.

A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, sep 2002. doi: 10.1613/jair.989.

Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009. doi: 10.1017/CBO9780511811357.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of IJCAI*, volume 7, pages 2462–2467, 2007.

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.

Samuel Dittmer and Igor Pak. Counting linear extensions of restricted posets, 2018.

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, 2015. doi: 10.1017/S1471068414000076.

Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, jun 1962. ISSN 0001-0782. doi: 10.1145/367766.368168.

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.

Timon Gehr, Sasa Misailovic, and Martin Vechev. Psi: Exact symbolic solver github, 2022. URL `https://github.com/eth-sri/psi/tree/master/test`.

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.

Zixin Huang, Saikat Dutta, and Sasa Misailovic. Aqua: Automated quantized inference for probabilistic programs. In *International Symposium on Automated Technology for Verification and Analysis*, pages 229–246. Springer, 2021.

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.

Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams: Learning with massive logical constraints. In *ICML Workshop on Learning Tractable Probabilistic Models (LTPM)*, June 2014.

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

D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.

D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

Jacob Laurel and Sasa Misailovic. Continualization of probabilistic programs with correction. In Peter Müller, editor, *Programming Languages and Systems*, pages 366–393, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44914-8.

D. H. Lehmer. Euclid's algorithm for large numbers. *The American Mathematical Monthly*, 45(4):227–233, 1938.

Alexander K Lew, Marco F Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K Mansinghka. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages*, 4(POPL): 1–32, 2019.

H.P. Luhn. Computer for verifying numbers. U.S. Patent US2950048A, 1960.

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.

Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD-foundations and applications*. Springer Verlag, 1998. doi: 10.1007/978-3-642-58940-9.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, pages 62–79. Springer, 2016. doi: 10.1007/978-3-319-29604-3_5.

Viktor Pfanschilling, Hikaru Shindo, Devendra Singh Dhami, and Kristian Kersting. Sum-product loop programming: From probabilistic circuits to loop programming. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 19, pages 453–462, 2022.

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

Feras Saad, Cameron Freer, Martin Rinard, and Vikash Mansinghka. The fast loaded dice roller: A near-optimal exact sampler for discrete probability distributions. In Silvia Chiappa and Roberto Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 1036–1046. PMLR, 26–28 Aug 2020.

Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. SPPL: probabilistic programming with fast exact symbolic inference. In *PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Design and Implementation*, pages 804–819, New York, NY, USA, 2021. ACM. doi: 10.1145/3453483.3454078.

Tian Sang, Paul Beame, and Henry A Kautz. Performing bayesian inference by weighted model counting. In *AAAI*, volume 5, pages 475–481, 2005.

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

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.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.

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.

Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. Swift: Compiled inference for probabilistic programming languages. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, page 3637–3645. AAAI Press, 2016. ISBN 9781577357704.

A PROOFS

A.1 BDD SIZES OF INTEGER DISTRIBUTION ENCODINGS (PROPOSITION 1)

A.1.1 Preliminaries

A b-rooted BDD B with m decision variables computes some function $\{0,1\}^m \to \{0,1\}^b$ at its roots. Note that we can treat each root of the BDD as corresponding to an individual function $\{0,1\}^m \to \{0,1\}$; in this manner we can equivalently treat such a BDD as a tuple of functions $(\{0,1\}^m \to \{0,1\})^b$. We will let $B(\vec{x}) = (B_1(\vec{x}), \ldots, B_b(\vec{x}))$ denote this tuple. Here, $\vec{x}$ contains the decision variables of the BDD, with the variable order of the BDD matching the order of the components of $\vec{x}$.

Note that the function computed by the BDD has as output a bit vector of length b. This can be interpreted as a b-bit unsigned integer, and we will treat the two interchangeably throughout these proofs. This also suggests that a function $f : \{0,1\}^m \to \{0, \ldots, 2^b - 1\}$ can equivalently specify a BDD; we will use $[\![f]\!]$ to denote the corresponding BDD.

We will be dealing with probabilistic BDDs, where each decision variable is given a weight representing a probability. Intuitively, this can be seen as replacing each x_i with a random $X_i = \mathrm{Bernoulli}(p_i)$; each decision variable has positive weight p_i and negative weight $1 - p_i$. The BDD can thus be viewed as representing a probability distribution over potential outputs, where the probability corresponds to the weighted paths through the BDD. We will say a probabilistic BDD $B(\vec{X})$ with probability assignments $\vec{p}$ *encodes* a probability distribution over unsigned integers $\Pr(V)$ if the two distributions are the same, i.e. the distribution over output bits of $(B(\vec{X}), \vec{p})$, when interpreted as integers, is identical to $\Pr(V)$.

A.1.2 Categorical Encoding of Integer Distributions

Recall Algorithm 1. Expanding the recursion and simplifying the arithmetic structure, we get the following natural encoding method for integers, which is similar to how the algorithm is implemented in practice.

$$g(\vec{x}) = \begin{cases} 0 & \text{if } x_0 \\ 1 & \text{else if } x_1 \\ \ldots & \\ 2^b - 2 & \text{else if } x_{2^b-2} \\ 2^b - 1 & \text{else} \end{cases}$$

In Algorithm 1, each x_i is given some probability p_i, corresponding to the flip in the algorithm. In this algorithm, we recurse on the tail of the input probability vector v, and so this probability $p_i = \frac{v[0]}{\sum v}$ represents the conditional probability that we return a number, given that our returned value is greater than or equal to that number. We formalize this encoding method in Definition 1.

Definition 1. Given a distribution $\Pr(V)$ over the integers $\{0, ..., 2^b - 1\}$, define $\mathrm{encode}_{\mathrm{CATEG}}(\Pr(V)) = ([\![g]\!], \vec{p})$, where $p_i = \Pr(V = i | V \geq i)$ for $0 \leq i \leq 2^b - 2$, and g is defined as follows:

$$g(\vec{x}) = \begin{cases} 0 & x_0 \\ 1 & \neg x_0 \wedge x_1 \\ \ldots & \\ j & x_j \wedge \bigwedge_{i=0}^{j-1} \neg x_i \text{ and } 0 \leq j \leq 2^b - 2 \\ \ldots & \\ 2^b - 1 & \bigwedge_{i=0}^{2^b-2} \neg x_i \end{cases}$$

Note that with the vector variable order $x_0, x_1, .., x_{2^b-2}$, this exactly matches the program variable order of Algorithm 1, and thus the evaluation order in the proposition.

Lemma 2. $\Pr(v \geq x) = \prod_{i=0}^{x-1} \Pr(v \neq i | v \geq i) \forall x, v \in \mathbb{N}_0$. *Proof omitted; proceed by induction.*

Lemma 3. *For any distribution* $\Pr(V)$ *over the integers* $\{0, ..., 2^b - 1\}$, $\text{encode}_{\text{CATEG}}(\Pr(V))$ *encodes* $\Pr(V)$.

Proof. Let $(\llbracket g \rrbracket, \vec{p}) = \text{encode}_{\text{CATEG}}(\Pr(V))$. We prove that for all $0 \le j \le 2^b - 1$, $\Pr(g(\vec{X}) = j) = \Pr(V = j)$.
Case 1: $0 \le j \le 2^b - 2$.

$$\begin{aligned}
\Pr(g(\vec{X}) = j) = \Pr\left(X_j \wedge \bigwedge_{i=0}^{j-1} \neg X_i\right) &= \Pr(X_j) \prod_{i=0}^{j-1} \Pr(\neg X_i) && \text{(Independence)} \\
&= \Pr(V = j | V \ge j) \prod_{i=0}^{j-1} \Pr(V \ne i \mid V \ge i) && \text{(As } \Pr(X_i) = p_i\text{)} \\
&= \Pr(V = j | V \ge j) \Pr(V \ge j) && \text{(Lemma 2)} \\
&= \Pr(V = j)
\end{aligned}$$

Case 2: $j = 2^b - 1$.

$$\begin{aligned}
\Pr(g(\vec{X}) = 2^b - 1) = \Pr\left(\bigwedge_{i=0}^{2^b-2} \neg X_i\right) &= \prod_{i=0}^{2^b-2} \Pr(\neg X_i) && \text{(Independence)} \\
&= \prod_{i=0}^{2^b-2} \Pr(V \ne i | V \ge i) && \text{(As } \Pr(X_i) = p_i\text{)} \\
&= \Pr(V \ge 2^b - 1) && \text{(Lemma 2)} \\
&= \Pr(V = 2^b - 1)
\end{aligned}$$

□

A.1.3 BDD Size of the CATEG_INT Encoding

We prove the first half of Proposition 1.

Proposition 1 (first half). A discrete distribution over the integers $\{0, 1 \ldots, 2^b - 1\}$ compiles to a BDD of size $\Theta(b2^b)$ when represented using CATEG_INT (Algorithm 1), with variables in flip evaluation order.

We will use $\text{LSB}_i(j)$ to denote the ith least significant bit of j; for example, $\text{LSB}_2(13)$ = 0.

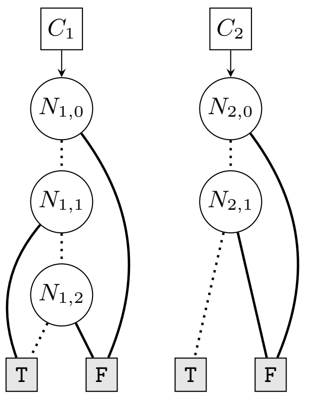
Lemma 4. *Let* $(\llbracket g \rrbracket, P) = \text{encode}_{\text{CATEG}}(\Pr(V))$, *where* $\Pr(V)$ *is the distribution over the integers* $\{0, .., 2^b - 1\}$. $\llbracket g \rrbracket$ *will be exactly the BDD C with the following structure:*

For each root $1 \le i \le b$, *C has nodes* $N_{i,0}, N_{i,1}, \ldots, N_{i,2^b-2^{i-1}-1}$, *at the levels of decision variables* $x_0, x_1, \ldots, x_{2^b-2^{i-1}-1}$, *respectively.*

$$\begin{aligned}
\text{low}(N_{i,j}) &= \textit{if} \quad j < 2^b - 2^{i-1} - 1 \quad \textit{then} \quad N_{i,j+1} \quad \textit{else} \quad 1 \\
\text{high}(N_{i,j}) &= \text{LSB}_i(j)
\end{aligned}$$

The roots of the BDD C_i *each point to the corresponding node* $N_{i,0}$.

Intuitively, the high edge of a node indicates that that decision variable is true, meaning that we have "chosen" our number (and thus the corresponding value for the bit); the low edge means that we should move on to the next bit. As an example, the BDD for $b = 2$ follows (terminal nodes visually duplicated for clarity).



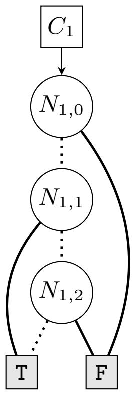
Proof. We argue that the described BDD is reduced. Note that the BDD essentially consists of linear chains from each root node. It is clear that no node has isomorphic children, as one child is always a terminal node while the other child is a decision node, and therefore we cannot eliminate any nodes. As the BDD consists of chains, merging nodes must be done between chains. As described, each chain is of a different depth, and so resolves on a different final decision variable. Therefore, at no stage can two nodes compute the same subfunction, and so we cannot merge nodes.

As our BDD is reduced, it is canonical for the represented function. Therefore, showing that the function represented by C is the same as g will show that our BDD $[\![g]\!]$ has the same structure.

We will now show that for all $\vec{x}$, $C_i(\vec{x}) = \mathrm{LSB}_i(g(\vec{x}))$, and thus that the functions are equivalent.

Let $S_{i,j}(\vec{x})$ denote the state of C_i at the level of x_j, given a (potentially partial) assignment to $\vec{x}$; this is the subfunction that must be computed at that point. This is usually a node at level x_j, but can be a lower level node such as a terminal node (representing that the decision node can be skipped). See the following BDD and corresponding states as an example.

$$\begin{aligned}
S_{1,0}(x_0,x_1,x_2) &= N_{1,0}\\
S_{1,1}(0,x_1,x_2) &= N_{1,1}\\
S_{1,1}(1,x_1,x_2) &= 0\\
S_{1,2}(0,0,x_2) &= N_{1,2}\\
S_{1,2}(0,1,x_2) &= 1\\
S_{1,2}(1,x_1,x_2) &= 0
\end{aligned}$$



Let $P(t)$ be the statement, $S_{i,t} = \text{if} \quad (g(\vec{x}) < t) \vee (t > 2^b - 2^{i-1} - 1) \quad \text{then} \quad \mathrm{LSB}_i(g(\vec{x})) \quad \text{else} \quad N_{i,t}$.

$P(1)$ holds as the initial node of all $C_i(\vec{x})$ is always $N_{i,0}$.

Assume $P(t)$ for an arbitrary $t \leq 2^b - 1$.

Consider going to the next node. If we are at a terminal node, then the state stays the same, otherwise we go to the low or high edge based on x_t. Thus, to advance state, we replace $N_{i,t}$ with if $\quad x_t \quad$ then $\quad \mathrm{high}(N_{i,t}) \quad$ else $\quad \mathrm{low}(N_{i,t})$.

$$\begin{aligned}
S_{i,t+1} = \text{if} \quad & (g(\vec{x}) < t) \vee (t > 2^b - 2^{i-1} - 1)\\
\text{then} \quad & \mathrm{LSB}_i(g(\vec{x}))\\
\text{else} \quad & (\text{if} \quad x_t \quad \text{then} \quad \mathrm{high}(N_{i,t}) \quad \text{else} \quad \mathrm{low}(N_{i,t}))
\end{aligned}$$

We substitute for $\mathrm{high}(N_{i,t})$ and $\mathrm{low}(N_{i,t})$.

$$S_{i,t+1} = \text{if} \quad (g(\vec{x}) < t) \vee (t > 2^b - 2^{i-1} - 1)$$

$$\begin{aligned}&\text{then} \quad \mathrm{LSB}_i(g(\vec{x}))\\ &\text{else} \quad \big(\text{if} \quad x_t \quad \text{then} \quad \mathrm{LSB}_i(t) \quad \text{else} \quad \big(\text{if} \quad t < 2^b - 2^{i-1} - 1 \quad \text{then} \quad N_{i,t+1} \quad \text{else} \quad 1\big)\big)\end{aligned}$$

$x_t \wedge \neg(g(\vec{x}) < t)$ implies $g(\vec{x}) = t$.

$$\begin{aligned}S_{i,t+1} = \text{if} \quad & (g(\vec{x}) < t) \vee (t > 2^b - 2^{i-1} - 1)\\ \text{then} \quad & \mathrm{LSB}_i(g(\vec{x}))\\ \text{else} \quad & \big(\text{if} \quad x_t \quad \text{then} \quad \mathrm{LSB}_i(g(\vec{x})) \quad \text{else} \quad \big(\text{if} \quad t < 2^b - 2^{i-1} - 1 \quad \text{then} \quad N_{i,t+1} \quad \text{else} \quad 1\big)\big)\end{aligned}$$

Two branches are now equivalent ($\mathrm{LSB}_i(g(\vec{x}))$) and can be combined.

$$\begin{aligned}S_{i,t+1} = \text{if} \quad & (g(\vec{x}) < t) \vee (t > 2^b - 2^{i-1} - 1) \vee x_t\\ \text{then} \quad & \mathrm{LSB}_i(g(\vec{x}))\\ \text{else} \quad & \big(\text{if} \quad t < 2^b - 2^{i-1} - 1 \quad \text{then} \quad N_{i,t+1} \quad \text{else} \quad 1\big)\end{aligned}$$

$g(\vec{x}) < t \vee x_t$ is equivalent to $g(\vec{x}) < t + 1$.

$$\begin{aligned}S_{i,t+1} = \text{if} \quad & (g(\vec{x}) < t + 1) \vee (t > 2^b - 2^{i-1} - 1)\\ \text{then} \quad & \mathrm{LSB}_i(g(\vec{x}))\\ \text{else} \quad & \big(\text{if} \quad t < 2^b - 2^{i-1} - 1 \quad \text{then} \quad N_{i,t+1} \quad \text{else} \quad 1\big)\end{aligned}$$

In the outer else, $t \leq 2^b - 2^{i-1} - 1$ (by the condition). In the inner else, t is also $\geq 2^b - 2^{i-1} - 1$.

$$\begin{aligned}S_{i,t+1} = \text{if} \quad & (g(\vec{x}) < t + 1) \vee (t > 2^b - 2^{i-1} - 1)\\ \text{then} \quad & \mathrm{LSB}_i(g(\vec{x}))\\ \text{else} \quad & \big(\text{if} \quad t = 2^b - 2^{i-1} - 1 \quad \text{then} \quad N_{i,t+1} \quad \text{else} \quad 1\big)\end{aligned}$$

In the inner else, $g(\vec{x}) > t$ by the outer condition and $t = 2^b - 2^{i-1} - 1$ by the inner condition; together, $g(\vec{x}) > 2^b - 2^{i-1} - 1$. For all such $g(\vec{x})$, $\mathrm{LSB}_i(g(\vec{x})) = 1$ (as $2^b - 1 - 2^{i-1}$) is the largest number at most $2^b - 1$ with the i^{th} bit set to 0). Thus, we can merge two more branches.

$$\begin{aligned}S_{i,t+1} = \text{if} \quad & (g(\vec{x}) < t + 1) \vee (t + 1 > 2^b - 2^{i-1} - 1)\\ \text{then} \quad & \mathrm{LSB}_i(g(\vec{x}))\\ \text{else} \quad & N_{i,t+1}\end{aligned}$$

Thus $P(t) \rightarrow P(t+1)$ and thus $P(t)$ holds for all $1 \leq t \leq 2^b - 1$.

Consider $P(2^b - 1)$.

$$\begin{aligned}S_{i,2^b-1} &= \text{if} \quad (g(\vec{x}) < 2^b - 1) \vee (2^b - 1 > 2^b - 2^{i-1} - 1) \quad \text{then} \quad \mathrm{LSB}_i(g(\vec{x})) \quad \text{else} \quad N_{i,2^b}\\ S_{i,2^b-1} &= \mathrm{LSB}_i(g(\vec{x}))\end{aligned}$$

Therefore $[\![g]\!]$ and C are equivalent. □

Proposition 1 (first half).

Proof. Let $(B, \vec{p}) = \mathrm{encode}_{\mathrm{CATEG}}(\Pr(V))$, where $\Pr(V)$ is a distribution over the integers $\{0, .., 2^b - 1\}$. It follows directly from Lemma 4 that $\sum_{i=1}^{b} 2^b - 2^{i-1} = b2^b - 2^b + 1$ decision nodes are needed for B.

□

A.1.4 Bitwise Encoding of Integer Distributions

Recall Algorithm 2. By expanding the algorithm, we obtain the following encoding method for integers, which again is similar to how it is implemented in practice.

$$\begin{aligned}
B_1(\vec{x}) &= x_\varepsilon \\
B_2(\vec{x}) &= \text{if } x_\varepsilon \text{ then } x_1 \text{ else } x_0 \\
B_3(\vec{x}) &= \text{if } x_\varepsilon \text{ then } (\text{if } x_1 \text{ then } x_{11} \text{ else } x_{10}) \text{ else } (\text{if } x_0 \text{ then } x_{01} \text{ else } x_{00}) \\
&\vdots
\end{aligned}$$

Our x_s are again derived from the algorithm; here, we recurse on the two halves of the input vector, and our probability p_s is the relative weight of the latter half of the vector (corresponding to the larger numbers, with a value of 1 for the MSB). While in the written algorithm we conditionally added a power-of-two, what we are in essence doing is probabilistically setting the value of the most significant bit. The subscript s refers to the sequence of more significant bits that have already been determined; for example, x_ε is deciding the value of the most significant bit (given an empty string, for no prior sequence), while x_{11} is deciding the third MSB, given that the first two MSBs are 11.

We formalize this encoding in Definition 2. Let s_i denote the i^{th} bit of a 1-indexed bit string or number; for example, $0010_3 = 1$. Let $|s|$ denote the length of a bit string. Let $\frown$ denote concatenation for bits and bit strings.

Note that we use two different types of subscripts: decision variables are subscripted by bit sequences for identification purposes, while a subscript i on values such as numbers and bitstrings represents the i^{th} bit of the value.

Definition 2. For a distribution $\Pr(V)$ over the integers $\{0,..,2^b-1\}$ with, define $\text{encode}_{\text{BITWISE\_INT}}(\Pr(V)) = (B, \vec{p})$ such that for each bit string s, $0 \le |s| < b$, there is a decision variable x_s with probability $p_s = \Pr(V_{|s|+1} \mid \bigwedge_{i=1}^{|s|} V = s_i)$.

Intuitively, the bits are chosen left-to-right, and each decision variable chooses the next bit given a bit string of past choices. Consider the following examples, where the empty bit string is denoted as ε.

$$\begin{aligned}
\Pr(x_\varepsilon) &= \Pr(V_1) \\
\Pr(x_0) &= \Pr(V_2 \mid \neg V_1) \\
\Pr(x_1) &= \Pr(V_2 \mid V_1) \\
\Pr(x_{0100}) &= \Pr(V_5 \mid \neg V_1 \wedge V_2 \wedge \neg V_3 \wedge \neg V_4)
\end{aligned}$$

We specify $B_i(\vec{x}) = x_{B_1(\vec{x})\frown B_2(\vec{x})\frown \ldots \frown B_{i-1}(\vec{x})}$ for $1 \le i \le b$. For example, $B_1(\vec{x}) = x_\varepsilon$, $B_2(\vec{x}) = x_{B_1(\vec{x})}$, $B_3(\vec{x}) = x_{B_1(\vec{x})\frown B_2(\vec{x})}$, and so on.

Lemma 5. *For any distribution* $\Pr(V)$ *over the integers* $\{0,..,2^b-1\}$, $\text{encode}_{\text{BITWISE\_INT}}(\Pr(V))$ *encodes* $\Pr(V)$.

Proof. Let $(B,\vec{p}) = \text{encode}_{\text{BITWISE\_INT}}(\Pr(V))$. We prove that for any possible assignment to bits $\vec{a} \in \{0,1\}^b$, $\Pr(B(\vec{X}) = \vec{a}) = \Pr(\vec{V} = \vec{a})$.

$$\begin{aligned}
\Pr(B(\vec{X}) = \vec{a}) &= \prod_{i=1}^{b} \Pr(B_i(\vec{X})) = a_i \mid \bigwedge_{j=1}^{i-1} B_j(\vec{X}) = a_j) && \text{(Chain rule of probability)} \\
&= \prod_{i=1}^{b} \Pr(X_{B_1(\vec{X})\frown \ldots \frown B_{i-1}(\vec{X})} = a_i \mid \bigwedge_{j=1}^{i-1} B_j(\vec{X}) = a_j) && \text{(Bit specification)} \\
&= \prod_{i=1}^{b} \Pr(X_{a_1\frown \ldots \frown a_{i-1}} = a_i \mid \bigwedge_{j=1}^{i-1} B_j(\vec{X}) = a_j) && \text{(Condition)} \\
&= \prod_{i=1}^{b} \Pr(X_{a_1\frown \ldots \frown a_{i-1}} = a_i \mid \bigwedge_{j=1}^{i-1} X_{B(\vec{X})_1\frown \ldots \frown B_{j-1}(\vec{X})} = a_j) && \text{(Bit specification)}
\end{aligned}$$

$$= \prod_{i=1}^{b} \Pr(X_{a_1^\frown \dots ^\frown a_{i-1}} = a_i) \qquad \text{(Independence)}$$

$$= \prod_{i=1}^{b} \Pr(V_i = a_i \mid \bigwedge_{j=1}^{i-1} V_j = a_j) \qquad \text{(As } \Pr(X_s) = p_s\text{)}$$

$$= \Pr(V = \vec{a}) \qquad \text{(Chain rule of probability)}$$

□

A.1.5 BDD Size of the BITWISE_INT encoding

We prove the second half of Proposition 1.

Proposition 1 (second half). A discrete distribution over the integers $\{0, 1 \dots, 2^b - 1\}$ compiles to a BDD of size $\Theta(2^b)$ when represented using BITWISE_INT (Algorithm 2), with variables in flip evaluation order.

Lemma 6. *Let* $\text{encode}_{\text{BITWISE\_INT}}(\Pr(V)) = (B, P)$*, where* $\Pr(V)$ *is the distribution over the integers* $\{0, ..., 2^b - 1\}$.

B will be exactly the BDD C with the following structure:

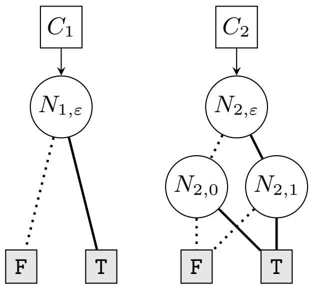
For each root $1 \leq i \leq b$, for all bit strings s of length less than i, let C have a node $N_{i,s}$ corresponding to decision variable x_s.

$$\begin{aligned} \text{low}(N_{i,s}) &= \textit{if} \quad |s| = i - 1 \quad \textit{then} \quad 0 \quad \textit{else} \quad N_{i,s^\frown 0} \\ \text{high}(N_{i,s}) &= \textit{if} \quad |s| = i - 1 \quad \textit{then} \quad 1 \quad \textit{else} \quad N_{i,s^\frown 1} \end{aligned}$$

The roots of the BDD C_i each point to the corresponding node $N_{i,\varepsilon}$.

Intuitively, for each root corresponding to bit i we consider up to a decision node of depth i (corresponding to a prefix bitstring of length $i - 1$); if we have not yet reached that depth, we instead go to a deeper decision node, with the appropriate new prefix.

As an example, the BDD for $b = 2$ follows (terminal nodes visually duplicated for clarity).



Proof. Note that the BDD described has variable order following the evaluation order from Algorithm 2; the direct descendant of a node corresponding to the decision variable x_s will either correspond to the decision variable $x_{s^\frown 1}$ or $x_{s^\frown 0}$, both of which are later in the evaluation order. As this holds for all nodes, the variable order must also follow globally.

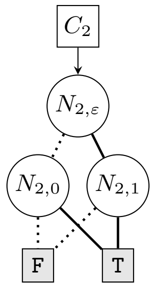
We use a similar argument to that in the proof of Lemma 4.

We first argue that the BDD is reduced. The argument is along the same line: each root now points to a tree deciding the value of the bit. No node has isomorphic children, as the two children correspond to either (necessarily different) terminal nodes, or decision nodes corresponding to different variables. In addition, as before each tree is of a different depth, corresponding to a different final decision variable, and so no two nodes on the same decision variable can compute the same subfunction. Therefore, we cannot merge nodes.

It remains to show that the function described by the BDD C is equivalent to that from the encoding B.

Let $S_{i,j}(\vec{x})$ denote the state of C_i at the level of the first decision variable whose bit string is of length j. See the following BDD and corresponding states as an example.

$$
\begin{aligned}
S_{2,0}(x_\varepsilon, x_0, x_1) &= N_{2,\varepsilon}\\
S_{2,1}(0, x_0, x_1) &= N_{2,0}\\
S_{2,1}(1, x_0, x_1) &= N_{2,1}\\
S_{2,2}(0, x_0, x_1) &= x_0\\
S_{2,2}(1, x_0, x_1) &= x_1
\end{aligned}
$$



Let $P(t)$ be the statement, $S_{i,t}(\vec{x}) = \text{if} \quad t \geq i \quad \text{then} \quad B_i(\vec{x}) \quad \text{else} \quad N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})}$.

$P(0)$ holds as there is no bit index greater than or equal to 0 and $S_{i,0}(\vec{x}) = N_{i,\varepsilon}$ by the placement of the roots.

Assume $P(t)$, which specifies $S_{i,t}$. Consider advancing to the next node in the BDD based on the assignment to decision variables (which does nothing if we are already at a terminal node):

$$
\begin{aligned}
\mathrm{next(S_{i,t})} = {} & \text{if} \quad t \geq i\\
& \text{then} \quad B_i(\vec{x})\\
& \text{else} \quad \big(\text{if} \quad x_{B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})} \quad \text{then} \quad \text{high}\big(N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})}\big) \quad \text{else} \quad \text{low}\big(N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})}\big)\big)
\end{aligned}
$$

We replace $x_{B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})}$ with $B_{t+1}(\vec{x})$.

$$
\begin{aligned}
\mathrm{next(S_{i,t})} = {} & \text{if} \quad t \geq i\\
& \text{then} \quad B_i(\vec{x})\\
& \text{else} \quad \big(\text{if} \quad B_{t+1}(\vec{x}) \quad \text{then} \quad \text{high}\big(N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})}\big) \quad \text{else} \quad \text{low}\big(N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})}\big)\big)
\end{aligned}
$$

We replace the high and low edges by the definition:

$$
\begin{aligned}
\mathrm{next(S_{i,t})} = {} & \text{if} \quad t \geq i\\
& \text{then} \quad B_i(\vec{x})\\
& \text{else} \quad (\qquad \text{if} \quad B_{t+1}(\vec{x})\\
& \qquad\qquad \text{then} \quad \big(\text{if} \quad t = i-1 \quad \text{then} \quad 1 \quad \text{else} \quad N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})^\frown 1}\big)\\
& \qquad\qquad \text{else} \quad \big(\text{if} \quad t = i-1 \quad \text{then} \quad 0 \quad \text{else} \quad N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})^\frown 0}\big)\big)
\end{aligned}
$$

We rearrange the if conditions:

$$
\begin{aligned}
\mathrm{next(S_{i,t})} = {} & \text{if} \quad t \geq i\\
& \text{then} \quad B_i(\vec{x})\\
& \text{else} \quad (\qquad \text{if} \quad t = i-1\\
& \qquad\qquad \text{then} \quad (\text{if} \quad B_{t+1}(\vec{x}) \quad \text{then} \quad 1 \quad \text{else} \quad 0)\\
& \qquad\qquad \text{else} \quad \big(\text{if} \quad B_{t+1}(\vec{x}) \quad \text{then} \quad N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})^\frown 1} \quad \text{else} \quad N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_t(\vec{x})^\frown 0}\big)\big)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{next(S_{i,t})} = {} & \text{if} \quad t \geq i\\
& \text{then} \quad B_i(\vec{x})\\
& \text{else} \quad (\qquad \text{if} \quad t = i-1 \quad \text{then} \quad B_{t+1}(\vec{x}) \quad \text{else} \quad N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_{t+1}(\vec{x})})
\end{aligned}
$$

The first two then branches collapse:

$$
\mathrm{next(S_{i,t})} = \text{if} \quad t+1 \geq i \quad \text{then} \quad B_i(\vec{x}) \quad \text{else} \quad N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_{t+1}(\vec{x})}
$$

As the only node we now reach has bit string length $t+1$, $\text{next}(S_{i,t}) = S_{i,t+1}$. Therefore $P(t) \to P(t+1)$.

Consider $P(b-1)$: $S_{i,b-1} = \text{if} \quad b-1 \geq i \quad \text{then} \quad B_i(\vec{x}) \quad \text{else} \quad N_{i,B_1(\vec{x})^\frown \ldots ^\frown B_{b-1}(\vec{x})}$.

For all $i \in \{1, \ldots, b-1\}$, we see that the root labeled $C_i(\vec{x})$ reaches the value $B_i(\vec{x})$. For $i = b$, the state reaches $N_{n,B_1(\vec{x})^\frown \ldots ^\frown B_{b-1}(\vec{x})}$, whose high and low edges are 1 and 0, respectively. The last step goes to $\big(\text{if} \quad x_{b_1^\frown \ldots ^\frown b_{b-1}} \quad \text{then} \quad 1 \quad \text{else} \quad 0\big) = B_b(\vec{x})$. Therefore B and C are equivalent. □

Proposition 1 (second half).

Proof. Let $\text{encode}_{\text{BITWISE\_INT}}(\Pr(V)) = (B, \vec{p})$, where $\Pr(V)$ is the distribution over the integers $\{0, ..., 2^b - 1\}$. It directly follows from Lemma 6 that B requires the following number of decision nodes.

$$\sum_{i=1}^{b} (\text{\# of bit strings of length less than } i) = 2^{b+1} - b - 2$$

□

A.2 ENCODING UNIFORM INTEGER DISTRIBUTIONS

Proposition 2. $\forall n > 0, \Pr(\text{UNIFORM}(n) = i) = \frac{1}{n}$ for $0 \leq i < n$.

Proof. We show this by strong induction on n. When $n = 1$ or $n = 2$, this result trivially holds. Consider $n = k$. Let $0 \leq i < k$, and $b = \lfloor \log_2(k) \rfloor$ as in the algorithm.

Case 1: $0 \leq i < 2^b$. $\text{UNIFORM}(k) = i$ requires us to take the first branch of the if statement. The conditional addition is equivalent to setting the b least significant bits of the number to 1 with probability $\frac{1}{2}$ and 0 with probability $\frac{1}{2}$. As each $0 \leq i < 2^b$ corresponds to a single unique binary string of length b, the probability of the bit sequence being equal to the wanted number is simply $(\frac{1}{2})^b$. Therefore, $\Pr(\text{UNIFORM}(k) = i) = (\frac{1}{2})^b(\frac{2^b}{k}) = \frac{1}{k}$.

Case 2: $2^b \leq i < k$. $\text{UNIFORM}(k) = i$ requires us to take the second branch of the if statement. Using our inductive hypothesis (as $k - 2^b < k$), $\Pr(\text{UNIFORM}(k) = i) = (\frac{k-2^b}{k}) \Pr(\text{UNIFORM}(k - 2^b) = i - 2^b) = (\frac{k-2^b}{k})(\frac{1}{k-2^b}) = \frac{1}{k}$, as wanted.

□

B EXPERIMENTS

Experimental Details All experiments were run on a server with a 2.20GHz CPU and 504GB RAM. WebPPL experiments were run on WebPPL v0.9.15; Psi experiments were run on Psi version `ec2cfc14a62a168afe7ce1d7269b92cf2882b830`. `Dice.jl` experiments were run using the code available at `https://github.com/Juice-jl/Dice.jl/tree/arithmetic`. The implementations of each model run are available at the same repository.

Benchmark Model Descriptions We provide a small description of our benchmarks in this section with details of their sources. The code implementations of all models are available at the repository `https://github.com/Juice-jl/Dice.jl/tree/arithmetic`.

1. book: A model of flipping towards a target page in a book adapted from the Psi test directory [Gehr et al., 2022].
2. tugofwar: Adapted from a traditional tug-of-war example [Huang et al., 2021], with values made discrete.
3. caesar: The caesar-cipher example from Dice [Holtzen et al., 2020], with a different number of characters being observed.
4. ranking: A model for learning a ranking system, adapted from Kisa et al. [2014].
5. radar1: A model of radar reception, adapted from a continuous model from Psi's [Gehr et al., 2016] benchmark suite.
6. floydwarshall: An implementation of the Floyd-Warshall algorithm [Floyd, 1962] on a graph with edges of random weight.

7. linear-extensions: A model counting linear extensions [Dittmer and Pak, 2018] where we observe a partial order and get an output distribution over all matching total orders.
8. triangle: A model categorizing a triangle of random side lengths, adapted from the Psi test directory [Gehr et al., 2022].
9. gcd: A model checking if two random numbers are coprime implementing Euclid's algorithm [Lehmer, 1938].
10. disease: A discrete disease model taken from existing works [Laurel and Misailovic, 2020].
11. luhn: A probabilistic model of student IDs leveraging the Luhn algorithm [Luhn, 1960].

Additional Experimental Results We provide additional experimental results supplementing those in the main paper.

BDD size serves as a proxy for how well knowledge compilation can exploit structure: the more compact the BDD, the smaller the representation of our function, and the faster weighted model counting can be executed. Table 2 compares the resultant BDD size for various models when compiled in Dice.jl using a binary or one-hot encoding. We can see that in almost all cases the binary encoding results in a smaller BDD, in some cases much smaller. Note that these models are those for which the one-hot encoding did not timeout; it is likely for those models that timed out that the true compiled BDD size will end up being much larger than the binary encoded BDD size.

Table 2: BDD sizes for probabilistic models using a binary vs one-hot encoding

Benchmarks	binary	one-hot
tugofwar	**2400**	2821
caesar-small	**1304**	3879
caesar-medium	**6344**	17879
caesar-large	**12644**	35379
ranking-small	**691**	1146
ranking-medium	**6218**	8297
ranking-large	**11491**	15680
radar1	**181**	332
floydwarshall-small	**10**	**10**
floydwarshall-medium	341	**237**
linear extensions-small	**29**	53
linear extensions-medium	**133**	257
linear extensions-large	**997**	1538
triangle-small	**10273**	150089
triangle-medium	**40419**	1156785
luhn-small	**518**	1010
luhn-medium	**2899**	7361

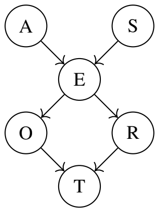
C BETA PRIOR APPLICATION: BAYESIAN NETWORK PARAMETER LEARNING

One natural application of a Beta prior is as a prior distribution for learning 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, there are some variables not given a value. In this setting, a Bayesian approach to parameter learning must consider all (exponentially many) possible instantiations of these missing values [Darwiche, 2009].

Using our Beta prior implementation described in the main paper, this setting can naturally be modeled within `Dice.jl`.. A Bayesian network can be expressed in the probabilistic program with Beta priors on each network parameter; a dataset can then be observed. By returning the distribution over our Beta parameters α and β, we obtain our posterior, a mixture over Beta distributions. These can then be manually combined to obtain the exact posterior density function.

We provide a brief example of this on the survey Bayesian network[2]. The structure of this network is shown in Figure 6; we simplify the network by making all variables binary so that the Beta is a suitable prior. We note that we can actually generalize to the non-binary case with a Dirichlet prior using an approach similar to that used for the Beta. We also provide an example missing dataset for the network; the the entry ? indicates a missing value. For this example, we focus on the specific parameter $\theta_{o|e} = \Pr(O = 1|E = 1)$, for which we give a uniform prior $\mathrm{Beta}(1, 1)$.

[2]https://www.bnlearn.com/bnrepository/



A	S	E	O	R	T
1	?	?	1	1	1
1	0	?	1	0	1
1	0	1	?	0	1
1	?	1	0	?	1
0	0	1	1	?	1
0	1	?	1	1	?
1	?	0	0	1	0
0	?	1	?	?	?
1	1	1	?	1	?
1	0	0	?	1	1

Figure 6: Visualization of and example missing dataset for the survey Bayesian network.

α	β	Pr(.)
9	3	0.226
8	4	0.207
10	2	0.177
7	5	0.160
6	6	0.109
5	7	0.066
4	8	0.035
3	9	0.016
2	10	0.005

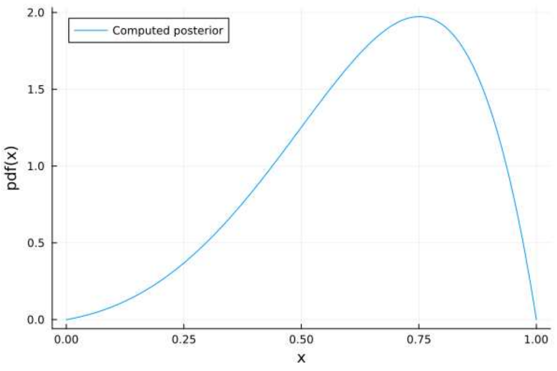


Figure 7: Output posterior distribution and visualization. The output represents a mixture over Beta distributions, represented by their parameters α and β.

By running the program representing this task, we get a large output distribution over Beta parameters - a mixture over Beta distributions. Note that it does not contain as many entries as possible instantiations, as some complete variable instantiations result in the same posterior Beta. If we plot the corresponding mixture, we can get a posterior PDF — note that this is an exact posterior recovered from the Beta parameters, in contrast to the approximate result one would get from sampling-based inference methods. The output and PDF visualization are given in Figure 7.

The code used for this example is available in the same `Dice.jl` repository (`https://github.com/Juice-jl/Dice.jl/tree/arithmetic`).