On Iteration in Discrete Probabilistic Programming

Mateo Torres-Ruiz
University College London, United Kingdom

Robin Piedeleu
University College London, United Kingdom

Alexandra Silva
Cornell University, United States of America

Fabio Zanasi
University College London, United Kingdom
University of Bologna, OLAS team (INRIA), Italy

Abstract

Discrete probabilistic programming languages provide an expressive tool for representing and reasoning about probabilistic models. These languages typically define the semantics of a program through its posterior distribution, obtained through exact inference techniques.

While the semantics of standard programming constructs in this context is well understood, there is a gap in extending these languages with tools to reason about the asymptotic behaviour of programs. In this paper, we introduce unbounded iteration in the context of a discrete probabilistic programming language, give it a semantics, and show how to compute it exactly. This allows us to express the stationary distribution of a probabilistic function while preserving the efficiency of exact inference techniques. We discuss the advantages and limitations of our approach, showcasing their practical utility by considering examples where bounded iteration poses a challenge due to the inherent difficulty of assessing the proximity of a distribution to its stationary point.

2012 ACM Subject Classification Theory of computation → Program semantics

Keywords and phrases Probabilistic programming, Programming languages semantics, Unbounded iteration

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Probabilistic Programming Languages (PPLs) combine the expressive power of classical programming constructs with the ability to draw random values from probability distributions and condition on the variables declared [17]. Unlike traditional programming languages, where evaluation leads to executing a routine, evaluating a probabilistic program results in computing the posterior probability distribution it specifies, a task called *inference*.

Inference can be either approximate [9, 16, 40], typically performed via sampling, or exact [7, 15, 31, 33], where the aim is to compute the *exact* posterior specified by the program. While most PPLs emphasise the use of continuous random variables, discrete random variables lend themselves to exact inference techniques. Moreover, they are better suited to several application domains, from cryptography [1, 29] to networks [13, 14, 38] or graphs [35]. These appealing properties have sparked renewed interest in discrete PPLs.

While finiteness of the discrete setting permits the computation of an exact posterior, exact inference is known to be undecidable for programs with an unbounded domain and #P-complete for programs which make use of Boolean variables only [36]. To tackle this difficulty, PPL compilers use data structures that exploit independence between variables, leading to efficient and scalable exact inference [21, 7].

Even though the semantics of probabilistic languages have been extensively studied ever since the seminal work of Kozen [26], and the standard programming constructs are well-

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

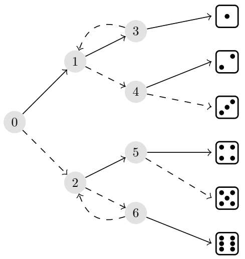
understood for discrete distributions, their extension to *unbounded iteration in the presence of exact inference* has been mostly absent. Possibly, due to the fact that unbounded iteration is seen to introduce non-termination, which can take the language semantics and inference capabilities outside the domain of discrete probabilities. As we show in this paper, this does not have to be the case: we extend a discrete PPL with unbounded iteration, while retaining the ability to perform exact inference.

Unbounded iteration is important to model probability distributions arising as the asymptotic behaviour of a (discrete-time) stochastic process, which are the natural object of study in a multitude of applications: given a ciphertext, we can construct a Markov chain to represent the posterior distribution of encryption keys [11]; in molecular physics, protein trajectories can be modeled using discrete-time Markov Chains (DTMCs), providing insights into the protein's biological function through their stationary distribution [34]; random walks on the symmetric group offer various applications such as diffusion models or shuffling schemes, where stationarity ensures reaching some optimal state [27]. From a probabilistic perspective, these can be seen as the outcome of iterating a program which models the one-step dynamics of the chain. Of course, one can always iterate such process a fixed number of times or until some condition is met. As we will see, this is often insufficient, and does not lead to the desired asymptotic behaviour in general. To reach the asymptotic limit, dedicated programming constructs for unbounded iteration are required.

The language we chose to extend is Dice [21], a functional PPL designed to handle discrete random variables and perform exact inference. Dice has a mature implementation and a very efficient exact inference engine. Before delving into our contributions, we provide a simple example that further motivates the use of unbounded iteration in probabilistic programming.

▶ **Example 1.** Consider the following probabilistic model which simulates a die using repeated throws of a coin c [25]. Starting at the root vertex, s_0, the model iteratively flips c and branches with a solid line whenever it lands heads, and with a dashed line when it results in tails. The coin-flipping procedure continues until a leaf is reached, i.e., until the value of the die is determined. We can write this as a probabilistic program:

```
fun knuth_yao(s) {
  if s == 0 then (if flip ½ then 1 else 2) else
  if s == 1 then (if flip ½ then 3 else 4) else
  if s == 2 then (if flip ½ then 5 else 6) else
  if s == 3 then (if flip ½ then ⚀ else 1) else
  if s == 4 then (if flip ½ then ⚁ else ⚂) else
  if s == 5 then (if flip ½ then ⚃ else ⚄) else
  if s == 6 then (if flip ½ then ⚅ else 2) else s }
```



The proposed model has 13 possible states: $\{0, \ldots, 6,$ ⚀$, \ldots,$ ⚅$\}$. The transition from the current state s is dependent upon both s and the result of a fair coin flip. For instance, starting from 0, we transition to either 1 or 2 based on the outcome of a coin flip, and it takes at least three consecutive runs to reach a ⚀, …, ⚅ state. Each function call yields not a single state but a *probability distribution* on the 13 potential states. While careful analysis of this and similar programs make it clear that *in the limit* the model indeed simulates a six-side fair die [23, 24], it is not immediately apparent that by iteratively applying this function, we will attain each possible die outcome with a probability of $\frac{1}{6}$.

Contributions. We extend Dice [21], a discrete PPL, with an unbounded iteration operator, give it a semantics that remains within the discrete realm, and show how to compute it exactly. Furthermore, we also consider an alternative form of iteration, which is more closely related to probabilistic while-loops and whose semantics is given by a least fixpoint.

We begin in Section 2 by formalizing the semantics of Dice. In Section 3 we highlight the difficulties encountered when attempting to use bounded iteration to obtain the asymptotic behaviour of some stochastic process. We then introduce the syntax and semantics for our proposed unbounded iteration operator in Section 4. An alternative form of iteration is then presented in Section 5, followed by a discussion of the implementation of both iterators, and an empirical evaluation of their performance in Section 6. Finally, we conclude with a discussion of related work and a few suggestions for future research in Section 7.

2 Dice: An Efficient Discrete PPL

In this section we review the syntax and semantics of Dice [21], a first-order non-recursive functional language featuring the essential characteristics of a discrete PPL. It supports Bernoulli distributions and has first-class observations which allow for Bayesian inference, while it offers an efficient approach to probabilistic inference by exploiting the internal representation of its programs.

2.1 Syntax

Dice supports Boolean and product types, as well as common branching and customary Boolean operators. The language features two probabilistic constructs: one for defining Bernoulli distributions (flip) and one for Bayesian evidence (observe). The first of these takes a real number as a parameter $\theta \in [0,1]$ and denotes a distribution that evaluates to true with probability θ and false with probability $1-\theta$, while the second takes an atomic expression a as a parameter and incorporates it as evidence that a evaluates to true throughout the program. This effectively changes the probability of all program paths where a does not hold to 0. The type of a Dice program is a product of Booleans, determined by the rules in Fig. 1.

A Dice program consists of a sequence of non-recursive functions followed by a main expression. We will later see that these expressions can be interpreted as discrete distributions on the declared values. Furthermore, it supports syntactic sugar for logical operations $(\wedge, \vee, \neg)$, bounded loops, bounded-size integers, and discrete distributions over k integers.

▶ **Example 2.** Consider the following straightforward setting: we have observed that trains to the nearest city experience delays with a 0.7 chance when it is raining and 0.3 otherwise. On any given day, there is a 0.1 probability of rain. If we observe that the trains are running late, what is the probability that it is raining? We can write down this in Dice as follows:

```
let x = flip 0.1 in
let y = if x then flip 0.7 else flip 0.3 in
let z = observe(y) in x
```

2.2 Semantics

Recall that a function $p : X \to [0,1]$ is a *probability distribution* whenever $\sum_{x \in X} p(x) = 1$, and is called a *subprobability distribution* when $\sum_{x \in X} p(x) < 1$. Closed Dice expressions can be viewed as a means of representing a subprobability distribution over the set of values attainable by the variables declared. The distributions expressible in Dice are furthermore *finitely supported*. That is, the set $\mathit{supp}(p) := \{x \in X \mid p(x) > 0\}$ is finite.

We can define well-typed expressions $\Gamma \vdash e \colon \tau$ and these will represent a (sub)probability distribution over the tuple of variables τ conditional on the value of the free variables that appear in Γ. To give a compositional interpretation of Dice expressions, we will interpret conditional distributions in a category of certain linear maps represented as matrices. In this sense, our presentation of the semantics of Dice is superficially different from that of the original paper, though the two are equivalent.

A nonnegative matrix is *stochastic* if each column sums to 1 and *substochastic* if the total of each column falls between 0 and 1. We write $\mathbb{R}^X$ for the vector space of real-valued functions on $\mathbb{R}$ which we equip with its Dirac basis $\{\delta_x \,|\, x \in X\}$ where $\delta_x(y) = 1$ if $x = y$ and 0 otherwise. Conditional (sub)distributions $p(y|x)$ can be thought of as linear maps $\varphi_p : \mathbb{R}^X \to \mathbb{R}^Y$ where φ_p is given by extending the mapping $x \mapsto \sum_y p(y|x)\delta_y$ linearly. The map φ_p then has the additional property that its matrix representation in the Dirac bases for $\mathbb{R}^X$ and $\mathbb{R}^Y$ is (sub)stochastic. We also call such maps (sub)stochastic maps. More generally, in what follows, we will speak of linear maps and matrices interchangeably, since all of our vector spaces are equipped with a chosen basis. We will use $\circ$ to denote composition of linear maps or simply juxtaposition to denote the product of matrices.

Since the product of substochastic matrices is substochastic, substochastic maps form a category Stoch with finite sets as objects and morphisms $X \to Y$ substochastic maps $\mathbb{R}^X \to \mathbb{R}^Y$. This is the category in which we interpret Dice programs. To interpret tuples, we also need to equip Stoch with the monoidal product given by the Cartesian product of sets on objects, and the usual tensor product on linear maps or, alternatively, the Kronecker product $\otimes$ on matrices. Note that the set of Diracs $\{\delta_x \otimes \delta_y \,|\, (x, y) \in X \times Y\}$ forms a basis of $\mathbb{R}^{X\times Y} \cong \mathbb{R}^X \otimes \mathbb{R}^Y$. This defines a monoidal structure over the category Stoch, which is moreover symmetric monoidal by equipping it with $\mathsf{swap}_{X,Y} : X \times Y \to Y \times X$ given by extending the mapping $\delta_x \otimes \delta_y \mapsto \delta_y \otimes \delta_x$ linearly.

▶ Remark 3. For the categorically-minded reader, the interpretation of Dice could also be given in the Kleisli category of the (sub)probability monad, which is equivalent to the category of (sub)stochastic linear maps. Therefore, most of the subsequent development could be phrased in this more categorical setting. We prefer a straightforward linear algebraic presentation, because our treatment of iteration in the coming sections makes use of standard results about Markov chains, which are usually formulated in a less abstract setting.

The type of the Booleans can be interpreted in Stoch as $[\![\mathbb{B}]\!] = \mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ and extends $[\![\cdot]\!]$ to arbitrary types and contexts $[\![()]\!] = 1 \cong \{0\}$, $[\![\tau \times \tau']\!] = [\![\tau]\!] \times [\![\tau']\!]$, $[\![\tau, \dots, \tau']\!] = [\![\tau]\!] \times \cdots \times [\![\tau']\!]$. To define the semantics of open terms, we will need the following stochastic maps for variable management.

- To discard a variable we will use the linear map dis_X obtained by extending $\delta_x \mapsto 1$ for all $x \in X$ linearly; for a single Boolean variable $\mathsf{dis}_{\mathbb{B}}$ is given by the stochastic matrix/row vector $\begin{pmatrix}1 & 1\end{pmatrix}$.
- To use a variable more than once, we use the linear map copy_X obtained by extending $\delta_x \mapsto \delta_x \otimes \delta_x$ linearly, for each $x \in X$; for a single Boolean variable $\mathsf{copy}_{\mathbb{B}}$ is represented by the stochastic matrix $\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}^\top$.

In general a typing statement $\Gamma \vdash t : \tau$ is interpreted as a substochastic map $[\![t]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$, where the semantics $[\![\cdot]\!]$ of an open term $\Gamma \vdash t : \tau$ is given recursively on the structure of the typing rules of Fig. 1.

- By convention, the Booleans are represented as the following column vectors: $[\![\mathsf{true}]\!] = \delta_{\mathsf{true}} = \begin{pmatrix}1 & 0\end{pmatrix}^\top$ and $[\![\texttt{false}]\!] = \delta_{\mathsf{false}} = \begin{pmatrix}0 & 1\end{pmatrix}^\top$.
- $[\![\texttt{flip}\ \theta]\!]$ is given by the Bernoulli distribution that assigns true with probability θ and false with probability $1 - \theta$. This is interpreted as the map $[\![\Gamma]\!] \xrightarrow{\mathsf{dis}_{[\![\Gamma]\!]}} 1 \xrightarrow{[\theta\ 1-\theta]^T} [\![\mathbb{B}]\!]$. Notice that $[\![\texttt{false}]\!]$ and $[\![\texttt{true}]\!]$ can be obtained as $\texttt{flip}\ \theta$ (in the empty context) where

$$\frac{}{\Gamma \vdash \texttt{true} : \mathbb{B}}\,(\text{true}) \qquad \frac{}{\Gamma \vdash \texttt{false} : \mathbb{B}}\,(\text{false}) \qquad \frac{}{\Gamma \vdash \texttt{flip}\ \theta : \mathbb{B}}\,(\text{flip})$$

$$\frac{\Gamma \vdash x : \mathbb{B}}{\Gamma \vdash \texttt{observe}\ x : \mathbb{B}}\,(\text{obs}) \qquad \frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}\,(\text{var}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\,(\text{tup})$$

$$\frac{\Gamma \vdash e : \tau_l \times \tau_r}{\Gamma \vdash \texttt{fst}\ e : \tau_l}\,(\pi_1) \qquad \frac{\Gamma \vdash e : \tau_l \times \tau_r}{\Gamma \vdash \texttt{snd}\ e : \tau_r}\,(\pi_2) \qquad \frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma, x : \tau_0 \vdash e_1 : \tau}{\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau}\,(\text{let})$$

$$\frac{\Gamma \vdash g : \mathbb{B} \qquad \Gamma \vdash e_T : \tau \qquad \Gamma \vdash e_F : \tau}{\Gamma \vdash \texttt{if}\ g\ \texttt{then}\ e_T\ \texttt{else}\ e_F : \tau}\,(\text{ite}) \qquad \frac{\Gamma, x \mapsto \tau \vdash e : \tau'}{\Gamma \vdash \texttt{fun}\ f(x : \tau) : \tau'\ \{e\} : \tau \to \tau'}\,(\text{fn})$$

$$\frac{\Gamma \vdash f : \tau \to \tau' \qquad \Gamma \vdash z : \tau}{\Gamma \vdash f(z) : \tau'}\,(\text{call})$$

Figure 1 Dice typing rules.

θ takes values 0 and 1, respectively.

- A variable $\llbracket x \rrbracket$ is interpreted by discarding the rest of the context:
$$\llbracket \Gamma, \tau, \Gamma' \rrbracket \cong \llbracket \Gamma \rrbracket \times \llbracket \tau \rrbracket \times \llbracket \Gamma' \rrbracket \xrightarrow{\mathsf{dis}_{\llbracket \Gamma \rrbracket} \otimes \mathsf{id}_{\llbracket \tau \rrbracket} \otimes \mathsf{dis}_{\llbracket \Gamma' \rrbracket}} 1 \times \llbracket \tau \rrbracket \times I \cong \llbracket \tau \rrbracket$$
- Variables are introduced through let-expressions, which are interpreted as the map
$$\llbracket \Gamma \rrbracket \xrightarrow{\mathsf{copy}_{\llbracket \Gamma \rrbracket}} \llbracket \Gamma \rrbracket \times \llbracket \Gamma \rrbracket \xrightarrow{\llbracket e_0 \rrbracket \otimes \mathsf{id}_{\llbracket \Gamma \rrbracket}} \llbracket \tau_0 \rrbracket \times \llbracket \Gamma \rrbracket \xrightarrow{\llbracket e_1 \rrbracket} \llbracket \tau \rrbracket$$
- Tuples (e_1, e_2) are interpreted as a joint distribution over the expressions involved, given by the map $\llbracket \Gamma \rrbracket \xrightarrow{\mathsf{copy}_{\llbracket \Gamma \rrbracket}} \llbracket \Gamma \rrbracket \times \llbracket \Gamma \rrbracket \xrightarrow{\llbracket e_1 \rrbracket \otimes \llbracket e_2 \rrbracket} \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket \cong \llbracket \tau_1 \times \tau_2 \rrbracket$.
- The projections $\llbracket \texttt{fst}\ e \rrbracket$ and $\llbracket \texttt{snd}\ e \rrbracket$ correspond to the marginalisation of joint distributions, given by applying $\mathsf{dis}_{\llbracket \tau \rrbracket}$ to the component over which we wish to marginalise, *i.e.*, $\llbracket \texttt{fst}\ e \rrbracket = \llbracket \Gamma \rrbracket \xrightarrow{\llbracket e \rrbracket} \llbracket \tau_l \rrbracket \times \llbracket \tau_r \rrbracket \xrightarrow{\mathsf{id}_{\llbracket \tau_l \rrbracket} \otimes \mathsf{dis}_{\llbracket \tau_r \rrbracket}} \llbracket \tau_l \rrbracket$ and $\llbracket \texttt{snd}\ e \rrbracket = \llbracket \Gamma \rrbracket \xrightarrow{\llbracket e \rrbracket} \llbracket \tau_l \rrbracket \times \llbracket \tau_r \rrbracket \xrightarrow{\mathsf{dis}_{\llbracket \tau_l \rrbracket} \otimes \mathsf{id}_{\llbracket \tau_r \rrbracket}} \llbracket \tau_r \rrbracket$.
- Conditionals $\llbracket \texttt{if}\ g\ \texttt{then}\ e_T\ \texttt{else}\ e_F \rrbracket$ take an expression g as guard whose truth value determines which of the branchings will be selected; its semantics is given by the map
$$\llbracket \Gamma \rrbracket \xrightarrow{\mathsf{copy}_{\llbracket \Gamma \rrbracket}} \llbracket \Gamma \rrbracket \times \llbracket \Gamma \rrbracket \xrightarrow{\mathsf{id}_{\llbracket \Gamma \rrbracket} \otimes \mathsf{copy}_{\llbracket \Gamma \rrbracket}} \llbracket \Gamma \rrbracket \times (\llbracket \Gamma \rrbracket \times \llbracket \Gamma \rrbracket) \xrightarrow{\llbracket g \rrbracket \otimes (\llbracket e_F \rrbracket \otimes \llbracket e_T \rrbracket)} \mathbb{B} \times (\llbracket \tau \rrbracket \times \llbracket \tau \rrbracket) \xrightarrow{\mathsf{join}_{\llbracket \tau \rrbracket}} \llbracket \tau \rrbracket$$
where $\mathsf{join}_X : \mathbb{B} \times X \times X \to X$ is the linear map obtained by extending the following mapping linearly: $\mathsf{join}_X(\delta_{\mathsf{true}}, x, y) = x$ and $\mathsf{join}_X(\delta_{\mathsf{false}}, x, y) = y$. For simple Boolean expressions, *i.e.*, for $\llbracket \tau \rrbracket = \mathbb{B}$, the corresponding 2-by-8 matrix is $\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$.
- Observe statements $\llbracket \texttt{observe}\ x \rrbracket$ incorporate evidence by assigning a 0 probability to executions where x does not hold. The interpretation of observations is given by the map $\mathbb{B} \xrightarrow{\mathsf{observe}} \mathbb{B}$, defined as the 2-by-2 matrix $\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$. Note that this is the only *sub*stochastic map which is not stochastic of all the Dice constructs.
- Functions are simply interpreted by keeping an environment mapping names to the semantics of the corresponding term: the (fn) rule then creates a new name for a given function, and (call) is interpreted as function application (note that Dice does not allow higher-order functions). This aspect of the semantics is not of much relevance to the development below, so we refer the interested reader to the original paper on Dice [21].

▶ **Example 4.** The semantics of the program in Example 2 is $(\mathsf{id}_{\mathbb{B}} \otimes (\mathsf{dis}_{\mathbb{B}} \circ \mathsf{observe})) \circ (\mathsf{id}_{\mathbb{B}} \otimes \mathsf{join}_{\mathbb{B}}) \circ (\mathsf{copy}_{\mathbb{B}} \otimes \mathsf{id}_{\mathbb{B}} \otimes \mathsf{id}_{\mathbb{B}}) \circ (\llbracket \mathsf{flip}\ 0.1 \rrbracket \otimes \llbracket \mathsf{flip}\ 0.7 \rrbracket \otimes \llbracket \mathsf{flip}\ 0.3 \rrbracket)$ which computes to $\begin{pmatrix} 0.07 \\ 0.27 \end{pmatrix}$.

Notice that the use of observe in the program makes this an unnormalised subprobability distribution. To interpret the outcome of a computation, the user needs a bona fide probability distribution. Hence, the Dice compiler automatically normalises the result at the end of a computation. From a denotational perspective, there is not much difference between the two, since one can always renormalise a nonzero subprobability. Note however that normalising is an expensive operation in principle. For this reason, the Dice compiler keeps track of the subprobability distribution that a given term denotes *and* of its associated normalisation constant as well. See Appendix A for a discussion of Dice's implementation details.

3 Iteration in PPLs: Existing Approaches and Challenges

It is straightforward to extend a discrete PPL with an operation for *bounded* iteration. In Dice, this problem was already addressed in [21] through the introduction of bounded loops, `iterate`(f, i, k), where f is a function name, i an initialization expression and k an integer specifying the number of times the function $f : \tau \to \tau$ should be iterated with a starting value i. While this allows for denoting a loop that terminates after a known number of iterations, it does not allow us to obtain the limiting behaviour of f. In this section, we review these limitations, thus justifying the need for *unbounded* iteration in general.

From Section 2.2, every Dice function of type $f : \tau \to \tau$ denotes a substochastic matrix and in turn, defines a DTMC. Conversely, every DTMC over a finite state-space can be encoded as a Dice program of type $\tau \to \tau$. The semantics of the Dice program f then defines a Markov chain on the state space $[\![\tau]\!]$. For a Markov chain with underlying transition matrix P, a vector π is a *stationary distribution vector* if $P\pi = \pi$. In many applications, Markov chains are useful when it can be shown that their behaviour closely approximates a given target within a reasonable number of steps in its time evolution. It is then useful to have a measure of how close a chain is from stationarity. For this purpose, the *mixing time* of a DTMC X with transition kernel P and stationary distribution π is often defined as $\min\{t : \max_{x\in\Omega} \|P^t(x,\cdot) - \pi\|_{TV} \leq 1/4\}$, where $\|\cdot\|_{TV}$ is the total variation distance. Intuitively, the mixing time gives the minimum amount of steps X must be iterated in order to be reasonably close to its limiting behaviour.

It may be tempting to sidestep the lack of an unbounded iteration operator by using `iterate`(f, i, k) for some sufficiently large k in order to approximate the stationary distribution π of some program f. However, determining the mixing time of an arbitrary expression f remains a challenging task. While classic Markov chain theory asserts that iterates of a chain with distribution P approach its stationary distribution subject to mild conditions on its internal structure (such as non-periodicity), it does not provide bounds on the number of steps required to be sufficiently close. This makes the naïve approach to iteration—using a bounded loop—unsatisfactory, as it does not allow the user to ascertain whether the posterior is close enough to stationarity. Or rather, such an approach offsets the burden of knowing the mixing time to the user, rather than the compiler.

Moreover, if empirical exploration of the distribution obtained by `iterate`(f, i, k) for different values of k can give an indication of progress towards stationarity, there are DTMCs where the convergence rate in total-variation undergoes an abrupt change over a negligible period of time [10]. This phenomenon, called *cutoff*, makes it infeasible to rely on heuristics based on statistical distances between successive iterations to determine when to stop the computation in the hope of getting a distribution that is sufficiently close to stationarity.

▶ **Example 5** (Limitations of finite iteration)**.** To illustrate the challenges in determining the stationary distribution of a DTMC, we examine the Bernoulli-Laplace model of diffusion.

This model can be depicted using two urns containing n balls each. Initially, the left and right urns contain n red balls and n blue balls, respectively. In each step, a ball is chosen at random from each urn, and they are exchanged. After sufficiently many exchanges, urns will be mixed, each of them containing approximately equal proportions of red and blue balls.

This process is fully determined by the number of blue (red) balls in one of the urns. Call this set $\Omega = \{1, \dots, n\}$. The transition matrix of this model can be built by considering how many blue balls will be in the left urn given that there are $b \leq n$ currently. There are three possibilities: either a blue ball or a red ball is grabbed from both urns ($P_{b,b} = 2\frac{b(n-b)}{n^2}$), a red ball is grabbed from the left urn and a blue ball is grabbed from the right urn ($P_{b,b+1} = \left(\frac{n-b}{n}\right)^2$), a blue ball is grabbed from the left urn and a red ball is grabbed from the right urn ($P_{b,b-1} = \left(\frac{b}{n}\right)^2$). This process can be encoded in the following Dice program:

```
fun bernoulli_laplace(blues_in_urn : int) {
  let urn =
  if blues_in_urn == 0 then (0, 1, 0, 0, ...)
  else if blues_in_urn == 1 then
     (1/n^2, (2*(n-1))/n^2, (n-1)^2/n^2, 0, ...)
  else if blues_in_urn == 2 then
     (0, 2^2/n^2, (2*(n-2)*2)/n^2, (n-2)^2, ...)
  ...
  else (..., 0, 0, 1, 0) in urn
}
let ith_trial = iterate(bernoulli_laplace, 0, i) in ith_trial
```

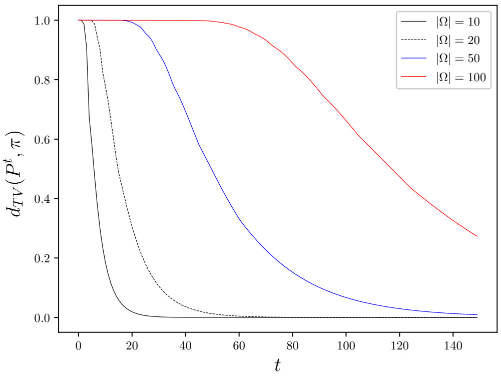
Note that this idealized encoding of the Bernoulli-Laplace model assumes a fixed urn size n, where "…" indicates a distribution that ought to follow above's model. Concrete instances of this program can be found in the tests provided in our implementation.

Consider the problem of determining the convergence of the above program to its stationary state. To assess the number of iterations required for the distribution to reach a sufficiently close approximation to the stationary state, we can conduct multiple experiments using various urn sizes. Analysis of the model [12] yields that for any finite state space Ω, for any $b \in \Omega$, the stationary distribution is given by

$$\pi(b) = \binom{n}{b}\binom{n}{n-b} \Big/ \binom{2n}{n}, \qquad 0 \leq b \leq n.$$

We can measure the time it takes to become reasonably close to this distribution by calculating the total-variation between the distribution of the urn at each iteration step and its stationary state for different urn sizes.

Note how the distance between the distribution and its stationary state remains close to its maximum value before dropping to near zero during a short period of time. This cutoff phenomenon shows the impracticality of approximate heuristics to determine the optimal number of iterations required for a probabilistic function to converge to a distribution that accurately represents its limiting behaviour.



Indeed, it has been proven that the Bernoulli-Laplace diffusion model exhibits a cutoff at time $\frac{1}{4}n \log n$ [12]. While a cutoff is not present in every DTMC, there are various families where it has been proven to exist [2, 10]. Furthermore, it remains a challenging task to determine if a chain has a cutoff window. For these reasons, a form of unbounded iteration is genuinely needed.

4 Introducing Unbounded Iteration

At first glance, it seems like unbounded iteration should force us out of the discrete probability setting by introducing nonterminating behaviour. How can potentially infinite execution runs be given a semantics with finitely-supported probability distributions? Perhaps surprisingly, we will now see that unbounded iteration can be accommodated in this setting.

If we think of a probabilistic program $t : \tau \to \tau$ as encoding the dynamics of some Markov chain, iterating t should give us the limiting behaviour of the chain, $\lim_{n\to\infty} [\![t]\!]^n$. However, this limit does not always exist, as is the case for example for chains that exhibit periodic behaviour. Thus, we cannot use it to define the semantics of unbounded iteration in general. Nonetheless, we can guarantee the existence of a limit by considering the *average time* spent on each of the possible states defined by the probabilistic function to be iterated: indeed, while the powers of an arbitrary stochastic matrix P need not have a limit, its *Cesàro sum* $\lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} P^k$ always exists [30, Section 8.4]. This is what we propose as semantics here. Note that this is not the first time that this is suggested as a reasonable semantics of iteration: in [8], the authors mention it in passing in the context of continuous probability. Here, we show that it gives a suitable semantics to iteration even in the discrete case. A precise way to compute it, together with a benchmark of our implementation against approximations that use bounded iteration are provided in Section 6.

Given a program $t : \tau \to \tau$, we want the program `iterate` t to denote a distribution that represents the average time spent on each element of $[\![\tau]\!]$ by all finite iterations of t. The typing rule for this iterator is given by

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash \texttt{iterate}\, t \ : \tau}$$

Operationally, we can think of `iterate` t as unfolding to the program $\mathsf{if}\ \mathsf{flip}\ \frac{1}{n}\ \mathsf{then}\ (\texttt{iterate}\ t)\ \mathsf{else}\ t^n$, assuming n is a runtime variable keeping track of the current execution step. Denotationally, the semantics of `iterate` t, is given by the Cesàro sum of $t : \tau \to \tau$, *i.e.*,

$$[\![\texttt{iterate}\ t]\!] := \left(\lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} [\![t]\!]^k \right)_0, \text{ where } A_0 \text{ denotes the first column of the matrix } A.$$

The choice of 1 here is arbitrary since, as we will see, all the columns of the Cesàro sum are equal. If we think of t as encoding the transition kernel of some Markov chain, this limit characterises the fraction of time the underlying Markov chain spends in each of its possible states, and its value is independent of the initial distribution we consider. The intuition is that the averaging of the Cesàro sum removes any oscillatory behaviour that could prevent the sequence of iterated powers to converge. The following example illustrates this point.

▶ **Example 6.** Consider the following program, one of the simplest programs exhibiting periodic behaviour (note that the exclamation mark denotes negation).

```
    fun swap_state(a : bool) : bool { !a }
    iterate swap_state
```

This program denotes a Markov chain that jumps between its only two states at every step. It is not hard to see that this chain has stationary distribution $\begin{pmatrix}1/2 & 1/2\end{pmatrix}^T$. Intuitively, if we distribute the total weight equally between the two states, then swapping them does not change the distribution. It should be noted that the periodic behaviour of the program makes it impossible to arrive at such distribution through successive iterations. Indeed, iterating the chain will simply shift the weight back and forth between the two states, while its Cesàro sum is equal to the stationary distribution.

4.1 Exact Inference

It is clear that to obtain $[\![\texttt{iterate } t]\!]$, we could simply compute $\frac{1}{n}\sum_{k=1}^{n}[\![t]\!]^k$ for successive values of n. However, the compiler would then need to decide when the approximation is sufficiently close to the limit to stop the computation. There are several possible heuristics to decide when a given approximation is satisfactory. For example, one could decide to stop when the distance between a given approximation and the previous one differ only by a small value ε in KL divergence. As we aim for exact inference, this approach would be insufficient, not to mention that it suffers from the same drawbacks as bounded iteration, highlighted in Section 3. It is in general a hard problem to determine whether the underlying Markov chain exhibits cutoff and when, thereby making it difficult to decide when we are reasonably close to the asymptotic behaviour of the chain. We can however sidestep these challenges by leveraging standard results from linear algebra that suggest a different–and most importantly, *exact*–way to compute the semantics of iteration, which we now explain.

Given any scalar sequence $(a_n)_{n\in\mathbb{N}}$, there is an associated *Cesàro* sequence of its averages, $(b_n)_{n\in\mathbb{N}}$, where $b_n = \frac{1}{n}\sum_{k=0}^{n-1} a_k$; we say that the sequence $(a_n)_{n\in\mathbb{N}}$ is *summable* to s (or just summable) whenever $\lim_{k\to\infty} b_k = s$. The same idea can easily be transported to the setting of matrices: given a sequence $(A^n)_{n\in\mathbb{N}}$ of powers of some square matrix A, we say that A is *convergent* when $\lim_{k\to\infty} A^k$ exists and that A is *summable* whenever its sequence of averages $\left(\frac{1}{n}\sum_{k=0}^{n-1} A^k\right)_{n\in\mathbb{N}}$ converges. Note that matrices that do not converge can be summable. Indeed, the averaging of a Cesàro sequence smooths away any periodic behaviour the chain could have. Furthermore, whenever a matrix A converges, its limit coincides with that of its Cesàro sequence [30]. In order for the Cesàro sum to be a suitable candidate for the semantics of $\texttt{iterate}\ t$, we must first make sure that this limit always exists for the family of matrices we are interested in. It is a well-known fact that this is always the case for (sub)stochastic matrices.

The following result about *substochastic* matrices is a mild generalisation of a theorem appearing in [30, Section 8.4] for stochastic ones. We will use it in Section 6, where we explain how our implementation computes the relevant projection and evaluate its performance.

▶ **Proposition 7.** *Every substochastic matrix M is summable and its sum is equal to the projection onto* $\mathsf{Ker}(\mathbf{I} - M)$ *along* $\mathsf{Im}(\mathbf{I} - M)$*.*

Interestingly, when the limit $\lim_n M^n$ exists, it is also equal to the projection onto $\mathsf{Ker}(\mathbf{I} - M)$ along $\mathsf{Im}(\mathbf{I} - M)$ and is therefore equal to the Cesàro sum [30, Section 7.10]. While we do not use this fact, it lends further support to the intuition that the Cesàro sum provides a generalised semantics for iteration, even in the context when the limit of finite iterations does not exist.

We make use of Proposition 7 when explaining how our implementation computes the relevant projection and evaluate its performance in Section 6.

5 An Alternative: Least Fixpoint Iterator

We can give an alternative form of iteration which is more closely related to imperative while-loops in the style of [26]. This follows the approach taken in [39], where iteration is defined by exploiting coproducts: starting from a program $t : A \to A + B$, $\mathsf{iterate}\ t\ \mathsf{from}\ u$ repeatedly calls t (starting from some value u) until the output of t is of type B, signalling the stopping condition for iteration, and returning the last output of t.

Although Dice does not have sum types, the same procedure can be mimicked through products and Booleans. We can represent the repeated application of some expression $t : \tau \to \mathbb{B} \times \tau$ that depends on an argument $x : \tau$ through an iterator that repeatedly calls t starting from some initial value u, in each step consuming a Boolean variable (that acts as a guard) until this variable evaluates to false, in which case the value of the last evaluation is output. The derived typing rule for this iterator is given by

$$\frac{\Gamma \vdash u : \tau \qquad \Gamma \vdash t : \tau \to \mathbb{B} \times \tau}{\Gamma \vdash \mathtt{iterate}_0\ t\ \mathsf{from}\ u : \tau}$$

The operational intuition given above suggests that $\mathtt{iterate}_0\ t\ \mathsf{from}\ u$ unfolds once to $\mathsf{let}\ (b, v) = t(u)\ \mathsf{in}\ (\mathsf{if}\ b\ \mathsf{then}\ (\mathtt{iterate}_0\ t\ \mathsf{from}\ v)\ \mathsf{else}\ v)$. In plain language, we apply t once to u, getting a pair (b, v) of a Boolean guard and a new value v; we then examine b to decide whether to apply t again or to return v.

We can think of this process in two ways: as iterating t from u or, equivalently, as iterating t and applying the resulting operation to u. With the latter perspective, we want to find a (sub)stochastic matrix M such that $M\llbracket u \rrbracket = \llbracket \mathtt{iterate}_0\ t\ \mathsf{from}\ u \rrbracket$. Then,

$$\begin{aligned}
M\llbracket u \rrbracket &= \llbracket \mathtt{iterate}_0\ t\ \mathsf{from}\ u \rrbracket \\
&= \llbracket \mathtt{let}\ (b, v) = t(u)\ \mathtt{in}\ (\mathtt{if}\ b\ \mathtt{then}\ (\mathtt{iterate}_0\ t\ \mathsf{from}\ v)\ \mathtt{else}\ v) \rrbracket \\
&= \mathsf{join}_\tau \circ (\mathsf{id}_\mathbb{B} \otimes M \otimes \mathsf{id}_\tau) \circ (\mathsf{id}_\mathbb{B} \otimes \mathsf{copy}_\tau) \circ \llbracket t \rrbracket \llbracket u \rrbracket
\end{aligned}$$

Since u can denote an arbitrary distribution, we are looking for a matrix M that is a solution of the equation above. This equation might have several solutions. However, in the presence of an order, it is traditional in programming language semantics to take the smallest solution as the canonical semantics of iteration [42]. For this purpose, we equip the set of substochastic maps with the pointwise order, *i.e.*, $A \le B$ for two substochastic matrices if $Ax \le Bx$ for all nonnegative $x \in \mathbb{R}^n$. With this order, we want the semantics of $\mathsf{iterate}_0\ t$ to be the least fixpoint of the map φ given by $\varphi(M) = \mathsf{join}_\tau \circ (\mathsf{id}_\mathbb{B} \otimes M \otimes \mathsf{id}_\tau) \circ (\mathsf{id}_\mathbb{B} \otimes \mathsf{copy}_\tau) \circ \llbracket t \rrbracket$ for $M : \llbracket \tau \rrbracket \to \llbracket \tau \rrbracket$. Note that $\mathsf{id}_\mathbb{B} = \llbracket \mathsf{true} \rrbracket \llbracket \mathsf{true} \rrbracket^T + \llbracket \mathsf{false} \rrbracket \llbracket \mathsf{false} \rrbracket^T$, so that

$$\begin{aligned}
\varphi(M) &= \mathsf{join}_\tau \circ \left(\left(\llbracket \mathsf{true} \rrbracket \llbracket \mathsf{true} \rrbracket^T + \llbracket \mathsf{false} \rrbracket \llbracket \mathsf{false} \rrbracket^T \right) \otimes M \otimes \mathsf{id}_\tau \right) \circ (\mathsf{id}_\mathbb{B} \otimes \mathsf{copy}_\tau) \circ \llbracket t \rrbracket \\
&= \mathsf{join}_\tau \circ \left(\left(\llbracket \mathsf{true} \rrbracket \llbracket \mathsf{true} \rrbracket^T \right) \otimes M \otimes \mathsf{id}_\tau \right) \circ (\mathsf{id}_\mathbb{B} \otimes \mathsf{copy}_\tau) \circ \llbracket t \rrbracket \\
&\quad + \mathsf{join}_\tau \circ \left(\left(\llbracket \mathsf{false} \rrbracket \llbracket \mathsf{false} \rrbracket^T \right) \otimes M \otimes \mathsf{id}_\tau \right) \circ (\mathsf{id}_\mathbb{B} \otimes \mathsf{copy}_\tau) \circ \llbracket t \rrbracket
\end{aligned}$$

Since $\mathsf{join}_X(\delta_\mathsf{true}, x, y) = x$ and $\mathsf{join}_X(\delta_\mathsf{false}, x, y) = y$, we can simplify the previous expression considerably: $\varphi(f) = M \circ (\llbracket \mathsf{true} \rrbracket^T \otimes \mathsf{id}_\tau) \circ \llbracket t \rrbracket + (\llbracket \mathsf{false} \rrbracket^T \otimes \mathsf{id}_\tau) \circ \llbracket t \rrbracket$. Thus, φ is an affine transformation over the vector space of linear maps $\mathbb{R}^{\llbracket \tau \rrbracket} \to \mathbb{R}^{\llbracket \tau \rrbracket}$ (which is isomorphic to $\mathbb{R}^{\llbracket \tau \rrbracket} \otimes \mathbb{R}^{\llbracket \tau \rrbracket}$), given by $\varphi(M) = MB + A$, where $A := (\llbracket \mathsf{false} \rrbracket^T \otimes \mathsf{id}_\tau) \circ \llbracket t \rrbracket$ and $B := (\llbracket \mathsf{true} \rrbracket^T \otimes \mathsf{id}_\tau) \circ \llbracket t \rrbracket$. Moreover, φ maps substochastic maps to substochastic maps, since $\varphi(M)$ can also be expressed as $\mathsf{join}_\tau \circ (\mathsf{id}_\mathbb{B} \otimes M \otimes \mathsf{id}_\tau) \circ (\mathsf{id}_\mathbb{B} \otimes \mathsf{copy}_\tau) \circ \llbracket t \rrbracket$, which is a composition of substochastic maps, and therefore substochastic.

In the literature [8, 26], the problem of characterising this fixpoint is usually approached by showing that φ is order-continuous—then, its least fixpoint can be obtained by taking the suprema of successive iteration of φ starting from the bottom of the order, *i.e.*, the uniform zero distribution:

$$\bigvee_{n\geq 0} \varphi^n(0) = \lim_{n\to\infty} \sum_{k=0}^{n} A \circ B^k$$

However, in general, this only gives a recipe to *approximate* the desired least fixpoint. In the discrete setting that is ours, we want to compute it *exactly*. We lay out the details on how we accomplish this to Section 6, where we elucidate the internal implementation of this iterator and evaluate its performance.

6 Implementation and Empirical Evaluation

We now extend the core syntax of Dice by incorporating the two iterators described in Sections 4 and 5 [41]. We evaluate their performance by comparing the average time required to attain the stationary distribution of a specific probabilistic function using these, contrasted with the time needed to approximate the same distribution through bounded iteration.

Dice is implemented in OCaml and uses rsdd [20], a Rust-based implementation for building and handling decision diagrams, as its backend for compiling binary decision diagrams (BDDs). To extend Dice, we employ py.ml [28], an OCaml library that facilitates the dynamic binding of Python modules, to harness efficient and extensively used linear algebra tools for the computation of linear operators corresponding to the semantics given by the iterators. While at compile time a Dice program is encoded as a weighted BDD, we obtain its matrix representation by evaluating its action on the canonical Dirac basis. This matrix is subsequently sent through py.ml to a Python library, where the distribution associated to either $\mathsf{iterate}$ or $\mathsf{iterate}_0$ is obtained. Subsequently, we obtain the weighted BDD representation of this distribution by leveraging Dice's internal encoding [21, Section 5.1] to define a distribution over a finite set of states. All our experiments were ran single-threaded on the same machine with a 2.7GHz CPU and 15GB of RAM. Timings were recorded using time, the standard GNU utility.

6.1 Cesàro Iterator

For computing the semantics of `iterate` t we work with the matrix representation of $[\![t]\!]$. From the discussion in Section 4 and Proposition 7, we know that the semantics of `iterate` t is any column of the projection onto $\mathsf{Ker}(\mathbf{I} - [\![t]\!])$ along $\mathsf{Im}(\mathbf{I} - [\![t]\!])$.

There are several ways to compute this projection. We have chosen to use a *full-rank factorisation* of $\mathbf{I} - [\![t]\!]$. A full-rank factorisation of a $n \times n$ matrix M is a pair of of an $n \times r$ matrix B and a $r \times n$ matrix C such that $M = BC$, where r is the rank of B, C, and M. Given such a factorisation, the projection onto $\mathsf{Im}\, M$ along $\mathsf{Ker}\, M$ is given by $P = B(CB)^{-1}C$, a result which can be found in § 7.10 of [30]. Since $\mathbf{I} - P$ is the complementary projector of P, the projection onto $\mathsf{Ker}\, M$ along $\mathsf{Im}\, M$ is therefore $\mathbf{I} - B(CB)^{-1}C$.

In practice, we obtain a full rank factorisation from the singular value decomposition (SVD) of M, by taking $M = U \begin{pmatrix} \Sigma & 0 \\ 0 & 0 \end{pmatrix} V^T = (U_0|U_1) \begin{pmatrix} \Sigma & 0 \\ 0 & 0 \end{pmatrix} \left(\frac{V_0}{V_1}\right)$, making $(U_0\Sigma)V_0 = U_0(\Sigma V_0)$ a full-rank factorisation of M, where U_0 corresponds to the matrix made by the first n columns of U where n is the number of non-zero singular values of Σ and, similarly, V_0 corresponds to the first n rows of V. Thus, $[\![$`iterate` $t]\!]$ is (say) the first column of $\mathbf{I} - U_0(V_0U_0)^{-1}V_0$, where U_0, V_0 come from the full-rank factorisation of $\mathbf{I} - [\![t]\!]$ given by its SVD (note that the singular values are not needed, since they cancel out for these choices of $\mathbf{B} = U_0\Sigma$ and $C = V_0$).

▶ **Example 8.** Consider the transition kernel $[\![t]\!]$ of the program that simply swaps two states (Example 6). An SVD of $M = \mathbb{I} - [\![t]\!]$ gives $M = U\Sigma V^T$ where $U = V = \begin{pmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$ and $\Sigma = \begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix}$. Thus, $[\![\texttt{iterate}\ t]\!]$ is any column of $\mathbf{I} - U_0(V_0U_0)^{-1}V_0$, which gives the distribution $\begin{pmatrix} \frac{1}{2} & \frac{1}{2} \end{pmatrix}^\top$. This witnesses the fact that the program oscillates between its only two states, spending equal time on each. Note that this gives a stationary distribution of the corresponding Markov chain which cannot be obtained by successive iterations.

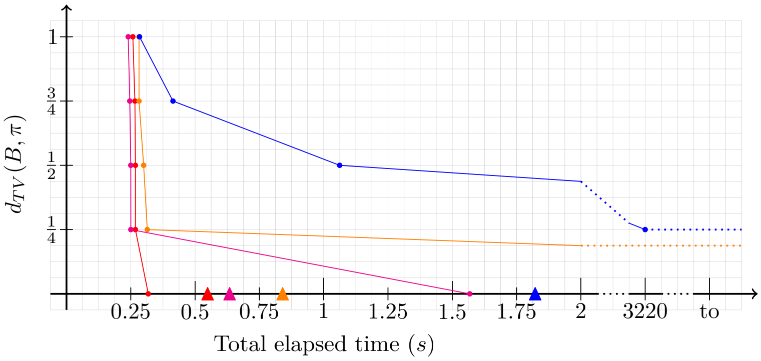


Figure 2 Comparative graph depicting the required time for various simulations of the Bernoulli-Laplace model to converge to a posterior distribution within $0+\epsilon, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}$ and 1 in total variation distance (drawn as points '•' in the above graph) from the stationary distribution, and the time it takes for $[\![\mathsf{iterate}\ t]\!]$ to compute the stationary distribution for an instance of the Bernoulli-Laplace model with equal number of states (drawn as triangles '▲'). Different colors represent distinct instances of the Bernoulli-Laplace model with different urn size. We extend the time line with a timeout data point (to) for the cases in which no posterior could be retrieved.

Empirical Evaluation We now evaluate the time performance of iterate. Note that a direct comparison of our iterative operator with other existing forms of iteration in Dice is not feasible. This is because while our iterator obtains the exact posterior distribution of some function $t : \tau \to \tau$, Dice can only approximate this distribution by simulating the underlying transition kernel of t by unfolding its BDD representation a fixed number of times.

To empirically evaluate the performace of $[\![\mathsf{iterate}]\!]$, we compare the time it takes for $[\![\mathsf{iterate}\ t]\!]$ to obtain the stationary distribution π for t an instance of the Bernoulli-Laplace diffusion model described in Section 3, with urns of various sizes $|\Omega| = \{3, 7, 10, 20\}$, and the time it takes for an iterative simulation to obtain a posterior distribution B at distances of $1, \frac{3}{4}, \frac{1}{2}, \frac{1}{4}$ and $0+\varepsilon$ in total variation from the stationary distribution π. Already for urns as small as $|\Omega| = 7$ it is not possible to obtain a posterior that is within $\varepsilon = 0.001$ distance from π in any reasonable amount time. Moreover, for models with an urn size of 10 or more, it becomes practically infeasible to achieve a posterior distribution beyond the mixing time of t. For this reason, we have decided to set $\epsilon = 0.005$ for the above experiments.

Colored dots in the graph in Fig. 2 represent the time it takes for the bounded iterative method to initially produce a posterior B that is within y-axis distance from π, while the colored triangles placed along the x-axis indicate the time required for $[\![\text{iterate } t]\!]$ to compute the exact posterior, π. It can be observed that even in experiments with small urn sizes, achieving a reasonably close approximation to the stationary distribution (reaching its mixing time) is not feasible through existing means.

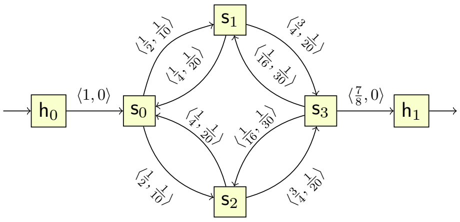
6.2 Least Fixpoint Iterator

As introduced in Section 5, the semantics of $\texttt{iterate}_0\ t$ from u is given by the fixpoint of equation (5). One way to compute this fixpoint *exactly* is with linear programming. Every fixpoint M of φ will satisfy $MA + B = M$ so we are looking for $M \in \mathbb{R}^{[\![\tau]\!]} \otimes \mathbb{R}^{[\![\tau]\!]}$ that satisfies $M(\mathbf{I} - A) = B$ and $M \geq 0$. We also need a linear form to minimise such that the solution to the corresponding linear program is the desired least fixpoint. In other words, we are looking for $C \in \mathbb{R}^{[\![\tau]\!]} \otimes \mathbb{R}^{[\![\tau]\!]}$ such that the minimum of $C^T M$ subject to the linear constraints $M(\mathbf{I} - A) = B$ and $M \geq 0$ is the least fixpoint of φ. Which C should we choose? We claim that any strictly positive C will work.

▶ **Proposition 9.** *Let C be any positive vector of $\mathbb{R}^{[\![\tau]\!]} \otimes \mathbb{R}^{[\![\tau]\!]}$. The least fixpoint of $\varphi : M \mapsto MA + B$ over the poset of substochastic maps is the solution to the following linear program: minimise $C^T M$ subject to the constraints $M(\mathbf{I} - A) = B$ and $M \geq 0$.*

Empirical Evaluation Now we compare the efficiency of $\text{iterate}_0\ t$ from u to obtain the least fixpoint of a probabilistic procedure against the time it takes to approximate the same distribution through finite iterative means. To this end, we adapt a probabilistic program that models a packet delivery reliability problem, inspired by Bayonet [14], a domain-specific language (DSL) focusing in network analysis, which captures the inherent probabilistic behaviour of computer networks. Similar to Dice, Bayonet allows to define discrete distributions within its program structure. This is achieved through PSI in its backend [15], a discrete PPL featuring a symbolic engine that generates concise expressions for representing distributions.

A network topology defines the network nodes which can be hosts or switches, and the links that interconnect them. Every link between two nodes is represented by two ports, and each of these nodes has both an input and an output queue, for received and outgoing packets, respectively. To capture the behaviour of a node, we specify how the node processes the packets it receives.



Consider above's network topology, where some switches s_i have a non-zero probability $\langle -, q \rangle$ of dropping a packet at any given moment and forwarding their packet with some probability $\langle p, - \rangle$. Host h_0 sends a packet to host h_1 every clock tick. Switch s_0 forwards half of h_0's packets to s_1 and the other half to s_2, with a $\frac{1}{10}$ chance of dropping a packet. Both s_1 and s_2 forward their packets to s_3 with a $\frac{3}{4}$ chance and send it back with a $\frac{1}{4}$ chance. At every moment, they have a $\frac{1}{20}$ chance of dropping the packet. Finally, s_3 has a $\frac{7}{8}$ chance of forwarding its packet to h_1 without any inconvenience and a $\frac{1}{8}$ chance of sending it back to either s_1 or s_2 with equal probability and a $\frac{1}{30}$ chance of dropping it.

The problem of packet delivery reliability is defined as the probability that a given packet (or flow of packets) reaches its intended destination. In our setting, we are interested in knowing the reliability of the entire network. This is, for any packet sent by h_0, what is the probability that it hits h_1. This network's behaviour is captured by the following program:

```
fun network(s_i) {
  if s_0 then (if flip 1/2 then (s_1, flip 1/10) else (s_2, flip 1/10)) else
  if s_1 then (if flip 3/4 then (s_3, flip 1/20) else (s_0, flip 1/20)) else
  if s_2 then (if flip 3/4 then (s_3, flip 1/20) else (s_0, flip 1/20)) else
  if flip 7/8 then (s_3, false) else
    (if flip 1/2 then (s_1, flip 1/30) else (s_2, flip 1/30))
}
fun aux(a) { let x = network(a) in (if (π_2 t) then (aux (π_1 t)) else (π_1 t)) }
```

The `network` function returns, for any state s, the distribution over the state space that represents the probability of transitioning to any other state. To evaluate the efficiency of $[\![\mathsf{iterate}_0\ t\ \mathsf{from}\ u]\!]$ in computing the fixpoint of the transition function implicitly given by the above program, we compare the time it takes to obtain this distribution via the least fixpoint iterator against the time it takes to *approximate it* through successive iterations, by unrolling the underlying BDD. Since, similar to the previous example, we cannot directly compare these two times, as one method finds the desired distribution exactly and the other only approximates it, we also provide how close we are in total variation distance to the target distribution π at each iteration i.

We denote by f^i the distribution given by the first projection of the output provided by $\mathtt{iterate}(\mathtt{aux}, s_0, i)$. Each column of the following table displays the average time taken by five different executions of f^i to obtain the posterior, along with its proximity to π, the stationary distribution of the transition given by `network`:

# iterations	1	3	5	7	9
$d_{TV}(f^i, \pi)$	0.912627	0.168776	0.023505	0.008805	0.013332
Ellapsed time (s)	0.252	0.266	0.296	1.030	58.32

While the simulation quickly converges to stationary, the oscillatory behaviour of the model hinders the possibility of establishing any stopping criterion. This naïve iterative method becomes unwieldy, already taking one minute to simulate only nine steps of the network. We contrast the times in the above table with the 0.54 seconds that it takes $\mathtt{iterate}_0$ $\mathtt{network}$ from s_0 to compute the stationary distribution.

7 Conclusion and Future Work

We have shown that it is possible to extend discrete probabilistic programming with unbounded iteration in at least two different ways, without giving up exact inference. Unbounded iteration is valuable because it allows us to compute the limiting behaviour of probabilistic programs that encode discrete-time stochastic processes. Since deciding when a stochastic process is sufficiently close to its limiting distribution is intractable in principle, we have argued that unbounded iteration and exact inference are both necessary in this setting. Finally, we have implemented these two different forms of iteration and demonstrated their performance on simple benchmarks.

Comparing the two iterators In this paper, we have given two forms of unbounded iteration: one based on averaging the behaviour of a term $t : \tau \to \tau$ over all iterations, and another given by iterating a term $t : \tau \to \mathbb{B} \times \tau$ from a given initial condition u, while the Boolean guard of t remains true. The former only makes sense in the context of probabilistic programming while the latter is a direct translation of while loops into the setting of a (functional) PPL.

While we have not given the Cesàro iterator as a fixpoint, it can also be characterised as one. Indeed, it is a well-known corollary of Proposition 7 that the Cesàro sum of a (sub)stochastic map M always gives a stationary distribution for the Markov chain it represents, in other words a fixpoint of M. Indeed, if P is the projection onto $\mathsf{Ker}(\mathbf{I} - M)$ along $\mathsf{Im}(\mathbf{I} - M)$, then for all x, $Px - MPx = 0$ and therefore $MPx = Px$.

At first sight, comparing these two forms of iteration is not meaningful, since they take arguments of different types. Therefore, one cannot simply ask when is $\texttt{iterate}\ t = \texttt{iterate}_0\ t\ \mathsf{from}\ u$, since the same t cannot be fed to both. Nevertheless, given $t : \tau \to \tau$, one can ask when does $\texttt{iterate}\ t = \texttt{iterate}_0\ (t, \texttt{true})\ \mathsf{from}\ u$, for some program u? It turns out that if the guard is always $\texttt{true}$, the semantics of $\texttt{iterate}_0\ (t, \texttt{true})\ \mathsf{from}\ u$ is uniformly 0 (regardless of u), which is not necessarily the case for $\texttt{iterate}\ t$. Conversely, given $t : \tau \to \mathbb{B} \times \tau$, one can obtain a program $t' : \tau \to \tau$ by marginalising over the Boolean guard. Even in this case, $\texttt{iterate}\ t'$ does not necessarily coincide with $\texttt{iterate}_0\ t\ \mathsf{from}\ u$. There are edge cases for which the two are equal (*e.g.*, when the spectral radius of t' is strictly less than 1, see Appendix A.2), but they do not correspond to any meaningful conditions on the corresponding program. Therefore, the forms of iteration have genuinely distinct capabilities and use cases: the first is useful to obtain the stationary distribution of some program encoding the dynamics of a Markov chain, while the second can be used to simulate while loops.

Related work. As far as we are aware, the only other line of work that combines exact inference and unbounded iteration is [6]. There, the semantics of programs is given in terms of probability generating functions and thus rather different from ours, making a direct comparison with our work more challenging. Moreover, while their language allows some infinitely-supported distributions, it does not allow conditioning inside of loops. In [7], the authors present an imperative PPL for discrete random variables, called BernoulliProb. Notably, BernoulliProb has while-loops and therefore support unbounded iteration. While the semantics of while-loops is closely related to the one we give to the least fixpoint iterator of Section 5, in practice, the implementation of [7] only approximates this least fixpoint using KL-divergence as a measure of convergence. Thus, their implementation suffers from the limitations we have highlighted in Section 3.

In [15], the authors introduce PSI, a PPL that is able to perform exact symbolic inference for programs with discrete and continuous random variables. However, the syntax of their language allows only bounded iteration. There are other general-purpose PPLs and frameworks including looping constructs with constant bounds [18, 32], leading to approximate rather than exact inference. These languages do not offer guarantees on the convergence rates of the associated fixpoint computation, running into the problems outlined in Section 3.

Finally, there are domain-specific PPLs that include some form of unbounded iteration [37, 38], but do so only for their restricted domain of application. In particular, the more narrow focus of [38] on network specification gives stronger guarantees on the convergence of the loops that can be specified in the language. This allows for exact inference in the presence of unbounded iteration, albeit in a setting which is less general than ours.

Future work. As mentioned in Remark 3, our work could be formulated in a more categorical context. This would enable us to connect the different notions of iteration and their corresponding fixpoints to traces in monoidal categories. In categories for which the monoidal product coincides with the categorical product, a trace always defines a notion of fixpoint. This is not necessarily the case in more general monoidal categories, such as that of (sub)stochastic maps. Finding a suitable trace would open the possibility of reasoning equationally about probabilistic programs with iteration.

References

1 Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. *SIGPLAN Not.*, 49(1):193–205, jan 2014. doi:10.1145/2578855.2535847.

2 Riddhipratim Basu, Jonathan Hermon, and Yuval Peres. Characterization of cutoff for reversible markov chains. *The Annals of Probability*, 45(3), may 2017. URL: https://doi.org/10.1214%2F16-aop1090, doi:10.1214/16-aop1090.

3 Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, sep 1992. doi:10.1145/136035.136043.

4 Dmitry Bugaychenko. On application of multi-rooted binary decision diagrams to probabilistic model checking. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, page 104–118, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-27940-9_8.

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

6 Mingshuai Chen, Joost-Pieter Katoen, Lutz Klinkenberg, and Tobias Winkler. Does a program yield the right distribution? verifying probabilistic programs via generating functions. In *International Conference on Computer Aided Verification*, pages 79–101. Springer, 2022.

7 Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 92–102, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491411.2491423.

8 Fredrik Dahlqvist, Alexandra Silva, and Dexter Kozen. *Semantics of Probabilistic Programming: A Gentle Introduction*, page 1–42. Cambridge University Press, 2020. doi:10.1017/9781108770750.002.

9 Swaraj Dash, Younesse Kaddar, Hugo Paquet, and Sam Staton. Affine monads and lazy structures for bayesian programming. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi:10.1145/3571239.

10 P Diaconis. The cutoff phenomenon in finite markov chains. *Proceedings of the National Academy of Sciences of the United States of America*, 93(4):1659—1664, February 1996. URL: https://europepmc.org/articles/PMC39998, doi:10.1073/pnas.93.4.1659.

11 Persi Diaconis. The markov chain monte carlo revolution. *Bulletin of the American Mathematical Society*, 46(2):179–205, November 2008. URL: http://dx.doi.org/10.1090/S0273-0979-08-01238-X, doi:10.1090/s0273-0979-08-01238-x.

12 Persi Diaconis and Mehrdad Shahshahan. Time to reach stationarity in the bernoulli-laplace diffusion model. *Siam Journal on Mathematical Analysis*, 18:208–218, 1987.

13 Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In Peter Thiemann, editor, *Programming Languages and Systems*, pages 282–309, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

14 Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. Bayonet: Probabilistic inference for networks. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 586–602, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192400.

15 Timon Gehr, Sasa Misailovic, and Martin Vechev. Psi: Exact symbolic inference for probabilistic programs. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 62–83, Cham, 2016. Springer International Publishing.

16 Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, UAI'08, page 220–229, Arlington, Virginia, USA, 2008. AUAI Press.

17 Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014. doi:10.1145/2593882.2593900.

18 Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. Probabilistic programming with densities in SlicStan: efficient, flexible, and deterministic. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, jan 2019. URL: https://doi.org/10.1145%2F3290348, doi:10.1145/3290348.

19 Holger Hermanns, Marta Kwiatkowska, Gethin Norman, David Parker, and Markus Siegle. On the use of mtbdds for performability analysis and verification of stochastic systems. *The Journal of Logic and Algebraic Programming*, 56(1):23–67, 2003. Probabilistic Techniques for the Design and Analysis of Systems. doi:10.1016/S1567-8326(02)00066-8.

20 Steven Holtzen. Rsdd. https://github.com/neuppl/rsdd/, 2023.

21 Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi:10.1145/3428208.

22 Roger A. Horn and Charles R. Johnson. *Matrix Analysis.* Cambridge University Press, 1985. doi:10.1017/CBO9780511810817.

23 Joe Hurd. Formal verification of probabilistic algorithms. Technical Report UCAM-CL-TR-566, University of Cambridge, Computer Laboratory, May 2003. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-566.pdf, doi:10.48456/tr-566.

24 Joost-Pieter Katoen. Model checking meets probability: A gentle introduction. In Manfred Broy, Doron A. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 177–205. IOS Press, 2013. doi:10.3233/978-1-61499-207-3-177.

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

26 Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981. doi:10.1016/0022-0000(81)90036-2.

27 D.A. Levin and Y. Peres. *Markov Chains and Mixing Times.* MBK. American Mathematical Society, 2017.

28 Thierry Martinez. pyml. https://opam.ocaml.org/packages/pyml/, 2023.

29 A.J. Menezes, J. Katz, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* Discrete Mathematics and Its Applications. CRC Press, 1996.

30 Carl D. Meyer. *Matrix Analysis and Applied Linear Algebra.* Society for Industrial and Applied Mathematics, USA, 2000.

31 T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

32 T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

33 Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in Hakaru (system description). In Oleg Kiselyov and Andy King, editors, *Functional and Logic Programming*, pages 62–79, Cham, 2016. Springer International Publishing.

34 Frank Noé. Probability distributions of molecular observables computed from Markov models. *The Journal of Chemical Physics*, 128(24):244103, 06 2008. arXiv:https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.2916718/15415898/244103_1_online.pdf, doi:10.1063/1.2916718.

35 Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Justin Chiu, Neeraj Pradhan, Alexander Rush, and Noah Goodman. Tensor variable elimination for plated factor graphs, 2019. arXiv:1902.03210.

36 Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1):273–302, 1996. doi:10.1016/0004-3702(94)00092-1.

37 Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. Cantor meets Scott: semantic foundations for probabilistic networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, jan 2017. URL: `https://doi.org/10.1145%2F3009837.3009843`, `doi:10.1145/3009837.3009843`.

38 Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. Scalable verification of probabilistic networks. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, jun 2019. URL: `https://doi.org/10.1145%2F3314221.3314639`, `doi:10.1145/3314221.3314639`.

39 Sam Staton. Commutative semantics for probabilistic programming. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 855–879, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

40 David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*, IFL 2016, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/3064899.3064910`.

41 Mateo Torres-Ruiz. Dice. `https://bitbucket.org/dice-iter/dice`, 2024.

42 Glynn Winskel. The formal semantics of programming languages - an introduction. In *Foundation of computing series*, 1993. URL: `https://api.semanticscholar.org/CorpusID:7767429`.

A Appendix

$\tau ::= \mathbb{B} \mid \tau_1 \times \tau_2$	(Types)
$v ::= \texttt{true} \mid \texttt{false} \mid (v, v)$	(Values)
$a ::= x \mid v$	(Atomic expressions)
$e ::= a \mid \texttt{fst}\ a \mid \texttt{snd}\ a \mid (a, a) \mid \textsf{let}\ x = e\ \textsf{in}\ e$	
$\mid \texttt{if}\ a\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{flip}\ \theta \mid \texttt{observe}\ a \mid f(e)$	(Expressions)
$\textsf{func} ::= \texttt{fun}\ f(x : \tau) : \tau\ \{e\}$	(Functions)
$p ::= e \mid \texttt{func}\ p$	(Programs)

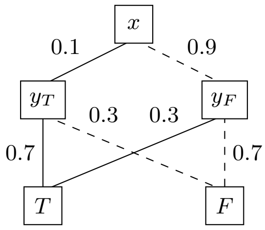
Table 1 Dice's core syntax. Metavariable f ranges over function names, x over variable names, and θ over reals in the range $[0, 1]$.

A.1 Core Dice

Dice programs are compiled into weighted Boolean formulas on top of which exact inference is performed via weighted model counting to obtain the distribution of densities specified in the main return expression [5]. To represent these formulas, the compiler internally manipulates programs by using multi-rooted binary decision diagrams (MRBDDs) [3], allowing it to take advantage of the conditional independence of the declared variables and the local structure amenable to efficient MRBDD compilation, making it possible for Dice to handle inference tasks even in the presence of models with a large number of variables.

Internally, the compiler maintains two Boolean formulas, φ and ψ. These represent all possible execution paths and only those that satisfy the Bayesian evidence provided by the observe statements, respectively. A global weighting function $w : L \to \mathbb{R}$ is employed to assign weights to literals. This allows for the assignment of weights to all models ω that satisfy a given formula, and the task of obtaining the posterior distribution of a given program is then reduced to a weighted model counting problem $\sum_{\omega \in \mathcal{M}(\varphi \wedge \psi)} w(\omega) / \sum_{\omega \in \mathcal{M}(\varphi)} w(\omega)$, where $\mathcal{M}(\alpha)$ is the set of models of some Boolean formula α.

▶ **Example 10.** The internal representation of the program in Example 2 is given by the following MRBDD, where the solid lines denote the paths where the decision variable is true and the dashed lines those where it is false.



where $\mathcal{M}(\varphi \wedge \psi) = \{x y_T\}$ and $\mathcal{M}(\varphi) = \{x y_T, \overline{x} y_F\}$, evaluating to $w(x y_T)/(w(x y_T) + w(\overline{x} y_F)) = (0.1 \cdot 0.7)/(0.1 \cdot 0.7 + 0.9 \cdot 0.3) \approx .2059$.

Dice can compile discrete probabilistic programs with bounded loops into a symbolic representation using MRBDDs by "unfolding" each iteration step. However, MRBDDs can only represent finite Boolean functions and thus are not suitable for directly encoding programs with unbounded loops. There is nevertheless a bijection between matrices and MRBDDs enabling us to move freely between the two representations as needed and leverage the diverse results from linear algebra to our advantage [4, 19].

▶ **Example 11.** The unfolding action on BDDs of bounded iteration leads to an increase in the number of nodes on its underlying representation, making it hard to reach sensible bounds on its approximation to the stationary distribution of the function that is iterated. Consider the problem of simulating a fair coin given samples from a biased one c with probability $p \in (0, 1)$ of landing heads. This can be done by employing von Neumann's trick: we flip c two times, if the outcomes are identical, we select the first sample as our sample, otherwise, we repeat the process. This procedure can be encode as the following program.

```
fun flip_biased_coin() {
  flip p
}

fun simulate_fair_coin(coin : bool) : (bool, bool) {
  snd (coin, coin ⊙ flip_biased_coin())
}

iterate(simulate_fair_coin, flip_biased_coin(), k)
```

While it is clear from the repeated runs of the above program that we converge to the expected value of $Bernoulli(\frac{1}{2})$, the number of nodes in its BDD representation grows linearly with the number of iterations k.

▶ Remark 12 (On normalisation). In general, the semantics of a term t might be a substochastic map. Nevertheless, eventually, the user would like to recover the normalised probability distribution encoded by a given program. While this is an expensive operation in general, the Dice compiler keeps track of an auxiliary formula, which encodes the paths through the compiled BDD allowed by the observe statements. This formula can be used to recover the normalisation constant and normalise $\llbracket t \rrbracket$ efficiently before computing $\llbracket \texttt{iterate}\ t \rrbracket$ or $\texttt{iterate}_0\ t\ \textsf{from}\ u$. In other words, we always normalise before iterating. The Cesàro sum of a stochastic map is a stochastic map, so $\llbracket \texttt{iterate}\ t \rrbracket$ will also be normalised. However, this is not always the case for $\texttt{iterate}_0\ t\ \textsf{from}\ u$: for example, if the the guard in $t : \tau \to \mathbb{B} \times \tau$ is always true, the semantics of the resulting iteration is uniformly zero. Note that this subprobability distribution cannot be normalised and interpreted as a genuine probability distribution. As we saw, this distribution is the bottom for the order over subprobability distributions and a program whose semantics assigns zero weights to all outcomes is better thought of as having an undefined value.

A.2 Least Fixpoint Iterator

Proof ot Proposition 9. Let us call M^* the least fixpoint of φ over the poset of substochastic maps. By definition, it satisfies $M^*(\mathbf{I} - A) = B$ and $M^* \geq 0$. Furthermore, if M is any other nonnegative fixpoint of φ, we have $M^* \leq X$, so that $C^T M^* \leq C^T M$. Indeed, this holds for any pair of vectors, since C has all positive entries. Thus, M^* is indeed the solution to the linear program given in the statement. ◀

▶ Remark 13. Whenever the spectral radius of B, the transition kernel of a Markov chain representing some program $t : \tau \to \tau$, is strictly bellow 1, there is a simpler method to compute the least fixpoint of φ than the outlined in Section 5. First, recall from 5 that

$$\bigvee_{n\geq 0} \varphi^n(0) = \lim_{n\to\infty} \sum_{k=0}^{n} A \circ B^k$$

And that for T an operator $\mathbb{R}^n \to \mathbb{R}^n$, its Neumann series $\mathbf{I}+T+T^2+\dots$ converges if and only if $\rho(T) < 1$ [22]. Moreover, in this case, $(\mathbf{I}-T)$ is invertible and $\sum_{n=0}^{\infty} T^n = (\mathbf{I}-T)^{-1}$. Indeed, if $(\mathbf{I}-T)$ was not invertible, there would be $u \in \mathbb{R}^n$ different from 0 such that $Tu = u$ and hence, $\rho(T) \geq 1$ since 1 would necessarily be an eigenvalue of T.

Therefore, when $\mathsf{Ker}(\mathbf{I}-B) = \{0\}$, we have $A(\mathbf{I}-B)^{-1} = A \displaystyle\lim_{n\to\infty} \sum_{k=0}^{n} B^k = \lim_{n\to\infty} \sum_{k=0}^{n} AB^k$, by continuity of matrix multiplication. We see that, in this case, $A(\mathbf{I}-B)^{-1}$ is the least fixpoint of φ. Note that whenever $\mathsf{Ker}(\mathbf{I}-B)$ is not trivial, the series $\sum_{n=0}^{\infty} B^n$ diverges and this approach fails.