Efficiently Sampling Probabilistic Programs via Program Analysis

Arun T. Chaganty
Stanford University

Aditya V. Nori
Microsoft Research India

Sriram K. Rajamani
Microsoft Research India

Abstract

Probabilistic programs are intuitive and succinct representations of complex probability distributions. A natural approach to performing inference over these programs is to execute them and compute statistics over the resulting samples. Indeed, this approach has been taken before in a number of probabilistic programming tools. In this paper, we address two key challenges of this paradigm: (i) ensuring samples are well distributed in the combinatorial space of the program, and (ii) efficiently generating samples with minimal rejection. We present a new sampling algorithm Qi that addresses these challenges using concepts from the field of program analysis. To solve the first challenge (getting diverse samples), we use a technique called symbolic execution to systematically explore all the paths in a program. In the case of programs with loops, we systematically explore all paths up to a given depth, and present theorems on error bounds on the estimates as a function of the path bounds used. To solve the second challenge (efficient samples with minimal rejection), we propagate observations backward through the program using the notion of Dijkstra's weakest preconditions and hoist these propagated conditions to condition elementary distributions during sampling. We present theorems explaining the mathematical properties of Qi, as well as empirical results from an implementation of the algorithm.

Appearing in Proceedings of the 16^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2013, Scottsdale, AZ, USA. Volume 31 of JMLR: W&CP 31.

1 INTRODUCTION

Probabilistic models, particularly those with causal dependencies, can be succinctly written as probabilistic programs. Recent years have seen a proliferation of languages for writing such probabilistic programs, as well as tools and techniques for performing inference over these programs (Gilks et al., 1994, Koller et al., 1997, Pfeffer, 2007a, Minka et al., 2009, Goodman et al., 2008, Kok et al., 2007, Gordon et al., 2013). Inference approaches can be broadly classified as static or dynamic. Static approaches compile the probabilistic program to a graphical model, and then perform inference over the graphical model (Koller et al., 1997, Minka et al., 2009, Kok et al., 2007) exploiting its structure. Dynamic approaches work by running the program several times using sampling to generate values, and perform inference by computing statistics over the results of several such runs (Pfeffer, 2007a, Goodman et al., 2008).

Dynamic approaches (which are also called sampling based approaches) are widely used, since running a probabilistic program is easy to perform, regardless of the programming language used to express the program. However, there are two main challenges with sampling based approaches. The first challenge is the quality and diversity of samples obtained from the joint probability distribution represented by the program. The main issue here is that there are many interdependent choices to be made during sampling, and choices that are unlikely apriori, may be highly likely aposteriori in light of observations or evidence. In the context of probabilistic programs, these choices correspond to exploring distinct paths in the program. Straightforward sampling of the program fails to sufficiently explore these paths, leading to poor estimated results. A second challenge for sampling from probabilistic programs (even along a single path) is that many samples that are generated during execution are ultimately rejected for not satisfying the observations. This is analogous to rejection sampling algorithms in various probabilistic models. In order to improve efficiency, it is desirable to avoid generating samples that are later rejected, to the extent possible.

The main contribution of this paper is a new sampling algorithm, called Qi[1], which uses program analysis techniques in order to address both the above challenges. Given a probabilistic program π as input, we first systematically decompose π into a sequence of feasible straight-line programs (corresponding to different paths in π), each of which can be sampled independently. The order in which these paths must be explored becomes crucial when the set of paths in the program becomes infinite, as in the case of probabilistic grammars or non-parametric models; we must ensure that the residual probability mass converges to zero. We prove that exploring paths roughly ordered by their depth guarantees this condition. In order to address the second challenge (avoiding many rejections during sampling), we propose augmenting sampling statements along each path (produced by the above path exploration procedure) using Dijkstra's weakest preconditions (Dijkstra, 1976) together with importance sampling weights in order to ensure that no samples are rejected. Informally, this corresponds to "hoisting" conditions on the joint distribution specified by a straight-line program to conditions on elementary distributions in the program. Together, these two techniques enable us to improve the quality and efficiency of sampling based estimation.

After computing estimates for each path of the program using sampling, we need to combine these estimates across paths by appropriately weighting samples along every feasible path π_i by the probability that the program takes the path π_i successfully (that is, the program takes all the branches in π, and satisfies all the observations along π_i). We present a scaling technique (see Section 5, Algorithm 3) by which we can estimate the probability that a program executes the path π_i successfully by estimating an appropriately defined indicator function. We show how to estimate the expected value of this indicator function by scaling the same samples obtained for estimating the result of the program.

We have implemented Qi and evaluated it on various benchmarks (see Section 6). Our empirical results are promising —our technique produces comparable estimates with the rejection sampling algorithm in Church (Goodman et al., 2008) with far fewer samples on all examples, and is able to produce more precise estimates in some examples.

Related work. There has been prior work on exploiting program structure to perform efficient sampling. Wingate et al. (2011) use nonstandard interpretation during runtime execution to compute derivatives, track provenance, and use these computations to improve the efficiency of MCMC sampling. Earlier work by Milch and Russell on BLOG (Milch and Russell, 2006) has used program structure to come up with good proposal distributions for MCMC sampling. Unlike these papers which use MCMC sampling, our work is based on importance sampling. Avi Pfeffer's work on general importance sampling (Pfeffer, 2007b) is closely related to our work. Our work improves upon Pfeffer's work in several ways. We make a detailed comparison with this work in Section 7.

```
bool earthquake, burglary, alarm,
phoneWorking, maryWakes, called;
earthquake = Bernoulli(0.0001);
burglary = Bernoulli(0.001);
alarm = earthquake or burglary;
if (earthquake)
  phoneWorking = Bernoulli(0.7);
else
  phoneWorking = Bernoulli(0.99);
if(alarm) {
  if(earthquake)
    maryWakes = Bernoulli(0.8);
  else
    maryWakes = Bernoulli(0.6);
} else
    maryWakes = Bernoulli(0.2);
called = maryWakes and phoneWorking;
observe(called);
return burglary;
```

Figure 1: Probabilistic program for Pearl's Burglar Alarm example.

2 OVERVIEW

We consider probabilistic programs written in a C-like imperative language equipped with two special statements to express probabilistic models:

1. The sampling statement allows sampling from standard distributions such as Bernoulli, Gaussian etc. For example, the statement "`x` = `Bernoulli`(0.4)" samples from a Bernoulli distribution with mean 0.4, and assigns the resulting value to variable `x`.

2. The observe statement allows conditioning the distribution with respect to an observation. For example, the statement "`observe`(`x` = true)" conditions the program to only consider executions where the value of variable `x` is true.

We allow all other statements of the C language such as conditionals, loops, function calls, pointers, arrays etc. Such programs represent probability distributions as in prior work (Koller et al., 1997, Pfeffer, 2007a, Goodman et al., 2008, Gordon et al., 2013).

[1]Qi = Quick Inference, and translates to "life force" in Chinese.

We explain our ideas using the probabilistic program shown in Figure 1. This program describes a joint probability distribution with 6 boolean variables: `earthquake`, `burglary`, `alarm`, `phoneWorking`, `maryWakes` and `called`. The return value of the program (see line 18) is the value of the variable `burglary`. Lines 2–16 specify how the 6 variables are assigned values, and the dependencies between these variables. The observe statement in line 17 conditions this distribution with the observed evidence that `called` is true.

Suppose we execute this program "as is" and perform sampling. We note that `earthquake` and `burglary` are true with very low probabilities and hence unlikely to be set to true during sampling. As a result several paths in the program above are likely to remain untraversed during sampling. Thus, the details of how the joint distribution is specified along these paths are ignored, resulting in inaccurate estimates for the value of the result produced by the program. Further, even along paths traversed frequently by the program, since line 17 requires the variable `called` to be true, several executions that set either `maryWakes` to false or set `phoneWorking` to false are filtered out since `called` is the conjunction of `maryWakes` and `phoneWorking`(see line 16).

Our idea behind using program analysis in Qi is two fold. Phase 1 of Qi replaces each probabilistic choice (i.e., the Bernoulli sampling statements in lines 2, 3, 6, 8, 11, 13 and 15) with nondeterministic choice and uses techniques from symbolic execution (Godefroid et al., 2005) to traverse all the 3 feasible paths in this program. The 3 paths are listed below as sequences of line numbers from the program:

$$
\begin{array}{ll}
\pi_1: & 2,3,4,5,6,9,10,11,16,17,18 \\
\pi_2: & 2,3,4,5,8,9,10,13,16,17,18 \\
\pi_3: & 2,3,4,5,8,9,15,16,17,18
\end{array}
$$

Note that simply traversing control flow paths in the program may result in infeasible paths. For instance, the path 2, 3, 4, 5, 6, 9, 10, 13, 16, 17, 18 is infeasible since the branches taken at lines 5 and 10 are inconsistent with each other. In Section 3, we show how to use symbolic execution (combined with concrete execution) to enumerate all the feasible paths of any probabilistic program, using a theorem prover (de Moura and Bjorner, 2008) (in other words, a logical inference engine) to rule out inconsistent or infeasible paths. In order to perform such path exploration in a scalable manner for large programs, we make use of advances in symbolic execution for test generation over the past decade (Godefroid et al., 2005, 2012, Cadar et al., 2008).

$$
\begin{array}{rcl}
wp(S_1;S_2,\phi) & = & wp(S_1, wp(S_2,\phi)) \\
wp(\text{observe } \psi,\phi) & = & \phi \wedge \psi \\
wp(x := e,\phi) & = & \phi[x := e] \\
wp(x \sim e,\phi) & = & \exists b.\phi[x := b]
\end{array}
$$

Figure 2: The $wp(S,\phi)$ computation. := denotes assignment and $\sim$ denotes sampling (or stochastic assignment).

Each of these paths can be thought of as straight-line programs. For instance path, π_1 corresponds to the program given below:

```
earthquake = Bernoulli(0.0001);
burglary = Bernoulli(0.001);
alarm = earthquake or burglary; observe(earthquake);
phoneWorking = Bernoulli(0.7); observe(alarm);
observe(earthquake); maryWakes = Bernoulli(0.8);
called = maryWakes and phoneWorking; observe(called);
return(burglary);
```

In phase 2 of Qi, we desire to generate samples that satisfy the observe statements in this straightline program. In order to do this, we push the predicates associated with observe statements back through the program toward every sampling statement using the technique of Dijkstra's weakest preconditions (Dijkstra, 1976). We then condition each sampling statement by its corresponding weakest precondition.

Doing this systematically involves weakest precondition computation (details in Section 5), and the result of such a computation for π_1 is shown in Table 1. For instance, the weakest precondition at line 17 is *true*, and since the statement at line 17 is "`observe(called)`", we have that the weakest precondition at line 16 is given by wp("`observe(called)`", true) = `called` (see Figure 2 for rules to compute wp). Once the weakest preconditions are calculated for each statement, we observe that as long as the sampling at each statement ℓ is done conditioned on its weakest precondition computed at ℓ, the generated sample is guaranteed to satisfy all the observe statements along the path. For instance, in our example, among the 3 sample statements at lines 2, 3 and 11, we have that the sample statements at lines 2 and 11 (which generate values for `earthquake` and `maryWakes` respectively) are conditioned to generate *true* values for these variables (since there are observe statements along this path that force these values to be *true*). On the other hand, the sample statement at line 3 (for generating the value of `burglary`) has the corresponding weakest precondition set to `earthquake`, which is independent of `burglary`. Thus, no conditioning is performed on this sample statement.

LINE#	STATEMENT	WEAKEST PRECONDITION AT LINE#
2	`earthquake` $=$ `Bernoulli`(0.0001)	`earthquake`
3	`burglary` $=$ `Bernoulli`(0.001)	`earthquake` $\wedge$ (`earthquake` $\vee$ `burglary`) $=$ `earthquake`
4	`alarm` $=$ `earthquake or burglary`	`earthquake` $\wedge$ `alarm`
5	`observe(earthquake)`	`earthquake` $\wedge$ `alarm`
6	`phoneWorking` $=$ `Bernoulli`(0.7)	`phoneWorking` $\wedge$ `earthquake` $\wedge$ `alarm`
9	`observe(alarm)`	`phoneWorking` $\wedge$ `earthquake`
10	`observe(earthquake)`	`phoneWorking`
11	`maryWakes` $=$ `Bernoulli`(0.8)	`maryWakes` $\wedge$ `phoneWorking`
16	`called` $=$ `maryWakes and phoneWorking`	`called`
17	`observe(called)`	*true*
18	`return(burglary)`	*true*

Table 1: Computation of weakest precondition for path π_1.

We note that once such conditioning is done, the above path program is equivalent to the program:

`burglary` $=$ `Bernoulli`(0.001); `return(burglary)`

Sampling this program results in an estimated value close to 0.001. Note that 0.001 is the expected value of `burglary` in path π_1, assuming all the observe statements and conditions are satisfied. Let us call this value y_1. Similarly, estimated values y_2 and y_3 can be calculated for each of the other paths π_2 and π_3 respectively. Qi combines the estimated values of each of the paths by weighting the estimated value y_i at each path π_i with a weight θ_i, where θ_i is the probability that the full program π executes path π_i and satisfies all the observe statements and conditionals. For example, the weight θ_1 associated with path π_1 is given by $0.0001 * 0.7 * 0.8 = 56 * 10^{-6}$. In Section 5, we show how to estimate the value of θ_i during the estimation of y_i, by sampling the program π_i.

3 THE Qi ALGORITHM

Algorithm 1 describes the Qi algorithm for efficiently sampling and performing inference on probabilistic programs. Qi takes a probabilistic program π as input together with two parameters κ_1 and κ_2, which are user-specified bounds on number of paths explored (useful in the case of programs with an infinite number of paths, or a large finite number of paths) and the number of samples used per path respectively. In line 1, Qi calls a procedure Explore that generates a sequence of straight-line programs Π, one for each path in P. Informally, Explore transforms P to a nondeterministic program (where all sample statements are replaced by nondeterministic assignments), and uses well-known path coverage techniques that combine concrete execution with symbolic execution in order to generate valid program paths (Godefroid et al., 2005). A *feasible path* is a path where there exists some value for all the variables that satisfies all the conditional and observe statements in it. Every such path is encoded as a straight-line program and added to the set Π. A precise description of Explore is given in Section 4. For every straight-line program $\pi_i \in \Pi$, the algorithm does the following (lines $3-6$). Every program $\pi_i \in \Pi$ is given as input to the procedure Estimate which generates samples from its posterior distribution by using Dijkstra's weakest conditions and likelihood weighting techniques. The procedure Estimate estimates the following two quantities.

Algorithm 1 Qi$(\pi,\ \kappa_1,\ \kappa_2)$

1: $\Pi :=$ Explore$(\pi,\ \kappa_1)$
2: $\Omega := \emptyset$
3: **for** $\pi_i \in \Pi$ **do**
4: $\quad(\theta, y) :=$ Estimate(π_i, κ_2)
5: $\quad\Omega := \Omega \cup \{(\theta, y)\}$
6: **end for**
7: **return** $\bar{\Omega}$

1. θ: the probability that executing the program results in the path π_i being exercised, and
2. y: the expected value returned by the program π_i conditioned on its path being exercised.

These estimates (θ, y) are accumulated in the set Ω. The details of the procedure Estimate are described in Section 5. Finally, Qi returns the weighted average $\bar{\Omega}$ (line 8) that computes the expectation of the value returned by the program. The weighted average $\bar{\Omega}$ is defined as follows.

$$\bar{\Omega} \stackrel{\text{def}}{=} \frac{\sum_{(\theta,y)\in\Omega}(\theta \times y)}{\sum_{(\theta,\_)\in\Omega}\theta}$$

It is important to note that the set of all paths of the input program π can be infinite in general, particularly in programs with unbounded loops and recursion. For instance, probabilistic programs modelling probabilistic context-free grammars (PCFG) may have an unbounded number of paths, each corresponding to a different parse tree. If Explore were to explore paths in

Algorithm 2 EXPLORE(π_i, κ)

1: $\pi_i^* := pp\_to\_np(\pi_i)$
2: $\Pi := \{\}$;
3: $F := \{\sigma_0\}$;
4: $d := d_0$;
5: **loop**
6:     $(C, F) :=$ EXECUTE(π_i^*, F, d)
7:     $\Pi := \Pi \cup C$;
8:     **if** $|\Pi| \geq \kappa$ **then**
9:         **break**
10:     **else**
11:         $d := d + \delta$;
12:     **end if**
13: **end loop**
14: **return** Π

such programs in a depth-first fashion, it would never consider some paths having a finite probability mass.

In the next section, we describe the EXPLORE procedure together with conditions that ensure that the probability mass of the programs in the tail of the sequence Π (as described in Algorithm 1) vanishes as we explore an increasing number of programs.

4 SYSTEMATIC PATH EXPLORATION

Let us first consider programs without loops (and hence a finite number of paths). For such programs, we can systematically enumerate all the control paths in the program by running dynamic programming algorithms on the control flow graph of the program. However, we desire to generate only paths that are *feasible* (as discussed in Sections 2 and 3). One way to do this is to perform symbolic execution along the path. Symbolic execution runs the program using a fresh symbolic value for every variable and generates path constraints (which are formulas) and check if the formulas are consistent using an automated theorem prover. Over the past decade, theorem provers that support SMT (Satisfiability Modulo Theories) such as Z3 (de Moura and Bjorner, 2008) have shown the ability to scale for large formulas.

In addition, we perform an optimization pioneered by DART (Godefroid et al., 2005) to cut down on the number of theorem prover calls, and scale symbolic execution to very large programs. The optimization works by simultaneously performing both symbolic and concrete execution along the program path, and using concrete values to make feasibility decisions, instead of invoking the theorem prover at every conditional.

Finally, in order to handle programs with loops (and hence an infinite number of paths), we use depth bounding and iteratively explore paths with larger depths until the number of paths explored exceeds the user supplied bound κ, as shown in Algorithm 2. The input probabilistic program π is first transformed into a nondeterministic program π^*, where all the sampling statements (probabilistic choice) are converted to nondeterministic choice (line 1). We maintain a "frontier" F, which is the set of incomplete paths we have explored so far, and a "depth bound" d, which is a bound on the length of the paths we want to explore. We initialize F to a path which contains the program counter of the first statement of π^*, denoted σ_0 (line 3), and d to an initial value $d_0 < \kappa$ (line 4). In the main loop of the algorithm (lines 5-13), we progressively increase d by δ (line 11), and invoke EXECUTE to explore paths starting from the current frontier F, bounded by depth d. The return value of EXECUTE is a pair (C, F), where C is a set of straight-line programs corresponding to the set of complete paths within the depth bound d, and F is the set of incomplete paths whose depths exceed d. We accumulate the set of paths C in the variable Π (line 7), until the cardinality of Π exceeds the user specified bound κ (line 8).

The following conditions on the EXPLORE procedure ensure that the probability mass of straight-line programs in the tail of the sequence Π vanishes as we explore an increasing number of programs.

Definition 1. EXPLORE *is a valid exploration procedure if the sequence of straight-line programs* Π *generated by it satisfy the following properties.*

1. *Each* $\pi_i \in \Pi$ *is unique.*
2. *For any* d*, there exists an* N*, such that for all* $n > N$*,* $|\pi_n| > d$*, where* $|\pi_n|$ *is the number of branches taken in the program* π *to generate* π_n*.*

Lemma 1. *Definition 1 is a sufficient condition for the sequence* $\{\pi_i\}$ *to have a probability mass* $\sum_{i=0}^{\infty} P(\pi_i)$ *that converges.*

Proof. Suppose that the exploration procedure did not satisfy the conditions of Definition 1. Then, there exists a d such that for every N, there is a $i > N$ with $|\pi_i| < d$. As $|\pi_i|$ measures the number of branch conditions taken, this quantity has a minimum of p^d, where p is the minimum branch probability, and is necessarily finite. Thus, for all N, there exists an $i > N$ such that $P(\pi_i) > p^d$; in other words, if the conditions in Definition 1 do not hold, $\sum_{i=0}^{\infty} P(\pi_i)$ does not converge by the limit test. ∎

It is easy to see that the EXPLORE procedure described in Algorithm 2 is a valid exploration procedure. With this definition of EXPLORE, we are able to prove the following theorem.

Theorem 1. *Let $\Pi = \{\pi_i\}$ be the sequence of straight-line programs returned by a valid exploration procedure on a probabilistic program π. Let x be the expression computed by π. Then, when $\mathbb{E}_\pi[x]$ exists, for every $\epsilon > 0$, there exists an N, such that for $n > N$, $|\mathbb{E}_\pi[x] - \sum_{i=1}^{n} P(\pi_i)\,\mathbb{E}_{\pi_i}[x]| < \epsilon$.*

Proof. For finite Π, $\mathbb{E}_\pi[x] = \sum_{i=1}^{|\Pi|} P(\pi_i)\,\mathbb{E}_{\pi_i}[x]$ by definition.

We now consider the case when Π is unbounded. $P(\pi_i)$ is simply the product of the probabilities of each branch taken along the path defined in π_i. Let p be the smallest branch probability; thus, $p^{|\pi_i|} < P(\pi_i) < (1-p)^{|\pi_i|}$. In other words, $P(\pi_i) \in O((1-p)^{|\pi_i|})$. Note also, that there are at most 2^d paths of depth d. In order for $\sum_i P(\pi_i)$ to converge, $p < \frac{1}{2}$ (i.e., p cannot be $\frac{1}{2}$). From the condition that $\mathbb{E}_\pi[x]$ is convergent, we know that $\lim_{i\to\infty} P(\pi_i)\,\mathbb{E}_{\pi_i}[x] \to 0$, or $\mathbb{E}_{\pi_i}[x] \in o(\frac{1}{P(\pi_i)})$.

Now, to show the condition of the theorem, we prove that the remainder, $\sum_{i=n+1}^{N} P(\pi_i)\,\mathbb{E}_{\pi_i}[x] < \epsilon$. To do so, let us first group together branches of equal depth,

$$
\begin{aligned}
\sum_{i=n+1}^{N} P(\pi_i)\,\mathbb{E}_{\pi_i}[x] &< \sum_{d=d'}^{\infty} 2^d O((1-p)^d) o(p^{-d}) \\
&< \sum_{d=d'}^{\infty} (2(1-p))^d \\
&< (2(1-p))^{d'} \frac{1}{1-2p}.
\end{aligned}
$$

If we choose an n such that $d' > \frac{\log((1-2p)\epsilon)}{\log(2(1-p))}$, then the inequality is guaranteed to hold. ☐

5 CONDITIONAL SAMPLING WITH WEAKEST PRECONDITIONS

In this section, we will focus on sampling from straight-line programs containing conditions and describe the ESTIMATE procedure (line 4 in Algorithm 1). The conventional approach to sampling from programs is rejection sampling (Pfeffer, 2007a, Goodman et al., 2008). Unfortunately, this approach can be prohibitively expensive, particularly when the observations are low probability events. Our main idea is to hoist observed conditions using Dijkstra's weakest preconditions (Dijkstra, 1976) in a straight-line program to the elementary probability distributions in it. Assuming that the elementary distributions can be sampled from efficiently given these conditions, we can guarantee that no sample is ever rejected. Algorithm 3 describes the ESTIMATE procedure for estimating θ and y (as described in Section 3) for an input straight-line program π_i. The call to *WP* in Line 2 computes the weakest precondition (defined formally in Figure 2) at every program point, and is maintained as a map τ from line number to the *wp* predicate. A property of this predicate is that every program state (assignment of variables to values) that satisfies it is guaranteed to satisfy all the subsequent observations in the straight-line program π_i. Therefore, it follows that making a random choice conditioned on the *wp* predicate at that point, will ensure that samples never get rejected. Table 1 illustrates the computation of the weakest precondition for the example straight-line program from Section 2. This entails pushing the predicate *true* from the last line of the program to the first line using the rules in Figure 2. We make two observations about weakest preconditions and the way we use them : (1) weakest preconditions are computed using substitutions and they are inexpensive to compute, and (2) weakest precondition computation only needs to happen once, irrespective of how many samples we wish to draw from the program.

Algorithm 3 ESTIMATE(π_i, κ)

1: $\Theta := \emptyset$, $\Omega := \emptyset$
2: $\tau :=$ *WP*(π_i, true)
3: **for** $j = 1$ to κ **do**
4: $\quad\alpha := 1.0$, $\beta := 1.0$
5: $\quad$**for** $l = 1$ to *lines*(π_i) **do**
6: $\quad\quad$**if** *stmt*(l, π_i) is $x \sim \mathcal{E}(\bar\theta)$ **then**
7: $\quad\quad\quad(w, x) :=$ *sample*$(\mathcal{E}(\bar\theta)|\tau[l])$
8: $\quad\quad\quad\alpha := \alpha \times w$
9: $\quad\quad\quad\beta := \beta \times \frac{P_{\mathcal{E}(\bar\theta)}(x)}{P_{\mathcal{E}(\bar\theta)|\tau[l]}(x)}$
10: $\quad\quad$**else**
11: $\quad\quad\quad$*execute*(π_i, l)
12: $\quad\quad$**end if**
13: $\quad$**end for**
14: $\quad\Omega := \Omega \cup \{(\alpha, \mathit{ret}(\pi_i))\}$
15: $\quad\Theta := \Theta \cup \{\beta\}$
16: **end for**
17: **return** $(\bar\Theta, \bar\Omega)$

Next, ESTIMATE iterates through a loop (lines 3–16) κ times (a parameter that defines the number of samples). Line 4 initializes to parameter α and β to 1.0. Lines 5–13 execute the program the following way. If the program statement at line number l is a probabilistic assignment $x \sim \mathcal{E}(\bar\theta)$ (lines 6–9), then x is drawn with the condition that it satisfies the *wp* predicate at that line number l (denoted by $\tau[l]$). This conditional sampling can be implemented by any importance sampling algorithm *sample* over the elementary distribution $\mathcal{E}(\theta)$. The result of such a sampling is the tuple (w, x) where w is the importance weight (which is equal to 1 if a true sample is drawn) and x is the variable being assigned to (line 7). The parameters α and β are updated on lines 8 and 9 respectively. On the other hand, if l is not a probabilistic assignment, then the statement at l is executed (line 11). Finally, the sets Ω and Θ accumulate the samples and the weighted average and average respectively over these sets is returned by ESTIMATE in line 17.

Theorem 2. *Let π_i be a straight-line probabilistic program with conditions φ and let κ be the sampling parameter. Then* ESTIMATE$(\pi_i,\ \kappa)$ *returns the correct estimates (over the k samples) of the conditional probability of $ret(\pi|\varphi)$ and the probability of the path associated with π_i.*

Proof. We use importance sampling to estimate $ret(\pi|\varphi)$ and the path probability. Each sample returned must be weighted by $\frac{P(x)}{Q(x)}$, where $P(x)$ is the true distribution, and $Q(x)$ is the proposal distribution from which the samples were drawn.

The w returned at each line $x \sim \mathcal{E}(\bar{\theta})$ correspond the weight $\frac{\mathcal{E}(\bar{\theta})}{Q(\bar{\theta})}$, where an appropriate proposal distribution Q may be chosen to draw the conditioned samples. Thus, the total weight associated with each sample will be $\frac{\pi_i(x)}{Q'(x)}$, where $Q'(x)$ is the combination of proposal distributions from which the program was sampled.

The computation of $P(\varphi)$ is taken as an expectation over all x. Thus, $P(\varphi) = \sum_x I[\varphi(x)]P(x) = \sum_{x:\varphi(x)} 1 \times P(x) + \sum_{x:\neg\varphi(x)} 0 \times P(x)$, where $I[\varphi(x)]$ is the indicator function taking value 1 iff x satisfies the condition φ. We exploit the fact that we only need to compute the expectation over the x that satisfy φ by using π_i itself as the proposal distribution. Thus, we require that the weights β be $E[\frac{P(x)}{P(x|\varphi)}]$, which is the quantity computed in lines 9 and 15 of Algorithm 3. □

6 EMPIRICAL EVALUATION

We evaluated our algorithm on several popular probabilistic programs and compared performance against the rejection sampling algorithm in Church (Goodman et al., 2008). A brief summary of the programs and their characteristics are presented in Table 2. We implemented both algorithms in the F# programming language. We used the theorem prover Z3 (de Moura and Bjorner, 2008) to check constraints during symbolic execution.

The results of experiments are summarized in Table 3. The results show that our QI algorithm is able to perform as well or better than Church's rejection sampling algorithm with far fewer samples (with corresponding reduction in execution time). The actual marginals (worked out analytically) are shown in the rows for algorithm ORACLE. For the Burglar Alarm example, QI was able to produce an exact solution. We also note that Red Light Game has an infinite number of paths. QI produces good results due to the path ordering heuristic from Section 3, however we would like to point out that the variance reported is inaccurate because it does not consider unexplored paths.

7 DISCUSSION

We presented a new algorithm QI which uses program analysis techniques to efficiently perform sampling on probabilistic programs. Our algorithm first considers the probabilistic choices as nondeterministic, and uses symbolic execution to generate feasible paths of the program. Then, along each path of the program, we hoist observations made backward using weakest precondition computation. We weight the estimated value computed along each path π_i with the probability that the path is executed and all the observations and conditions are satisfied, and combine all the weighted estimates from all paths. If the program has a large number or infinite number of paths, we show how to pick a fixed number of paths such that we can bound the error in estimation due to omitting paths.

Our work generalizes earlier work by Pfeffer (2007b) on importance sampling. Pfeffer's work presents several structural heuristics (such as conditional checking, delayed evaluation, evidence collection and targeted sampling) to help make choices during sampling that are less likely to get rejected by observations. The second phase of our algorithm unifies and generalizes all these heuristics using one concept – weakest preconditions. This enables us to handle not only all the examples in Pfeffer's paper using one technique, but also enables us to handle examples with predicates such as linear arithmetic, which are beyond the reach of Pfeffer's heuristics, but can be handled using weakest preconditions and theorem provers. Further, there are no analogs for path selection phase of our algorithm in Pfeffer's work. Using our symbolic execution, we are able to efficiently enumerate paths, and also handle recursive programs and programs with loops by carefully choosing the order in which we explore paths.

Acknowledgements

We thank Selva Samuel for very helpful comments on an earlier draft of this paper.

NAME	DESCRIPTION	REFERENCE
Grass Model	Small model relating the probability of rain, having observed a wet lawn.	(Kiselyov and Shan, 2009), (Goodman et al., 2008)
Burglar Alarm	Example given in Figure 2.	Adapted from Pearl
Noisy OR	Given a DAG, each node is a noisy-or of its parents. Find posterior marginal probability of a node, given observations	(Kiselyov and Shan, 2009)
Red Light Game	Planning-as-inference example in which the probability of winning the game given the first action is modeled. Notably, this program exhibits unbounded recursion.	(Goodman et al., 2008)

Table 2: Evaluated Programs.

NAME	ALGORITHM	SAMPLES (REJECTIONS)	ESTIMATED VALUE	TIME TAKEN(s)
Grass Model	Qi	600	$0.70107 \pm 1e-4$	1.1
Grass Model	Church	600 (940)	$0.70391 \pm 1e-4$	4.9
Grass Model	Oracle	-	0.7079	-
Burglar Alarm	Qi	30	0.0743 ± 0	1.0
Burglar Alarm	Church	200 (1925)	$0.0675 \pm 3e-4$	12.7
Burglar Alarm	Oracle	-	0.0743	-
Noisy OR	Qi	2000	$0.465 \pm 1e-4$	1.9
Noisy OR	Church	5000 (16573)	$0.463 \pm 3e-4$	84.3
Noisy OR	Oracle	-	0.4626	-
Red Light Game	Qi	200	0.7683 ± 0	7.1
Red Light Game	Church	200 (24732)	$0.5985 \pm 7e-4$	163.1
Red Light Game	Oracle	-	0.768	-

Table 3: Evaluation Results.

References

Cadar, C., Dunbar, D., and Engler, D. R. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation (OSDI)*, pages 209–224.

de Moura, L. and Bjorner, N. (2008). Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice Hall.

Gilks, W. R., Thomas, A., and Spiegelhalter, D. J. (1994). A language and program for complex Bayesian modelling. *The Statistician*, 43(1):169–177.

Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: Directed Automated Random Testing. In *Programming Language Design and Implementation (PLDI)*, pages 213–223.

Godefroid, P., Levin, M. Y., and Molnar, D. A. (2012). Sage: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44.

Goodman, N. D., Mansinghka, V. K., Roy, D. M., Bonawitz, K., and Tenenbaum, J. B. (2008). Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI)*, pages 220–229.

Gordon, A. D., Aizatulin, M., Johannes Borgstroem, G. C., Graepel, T., Nori, A. V., Rajamani, S. K., and Russo, C. (2013). A model-learner pattern for bayesian reasoning. In *Principles of Programming Languages (POPL)*.

Kiselyov, O. and Shan, C. (2009). Monolingual probabilistic programming using generalized coroutines. In *Uncertainty in Artificial Intelligence (UAI)*, pages 285–292.

Kok, S., Sumner, M., Richardson, M., Singla, P., Poon, H., Lowd, D., and Domingos, P. (2007). The Alchemy system for Statistical Relational AI. Technical report, University of Washington.

Koller, D., McAllester, D. A., and Pfeffer, A. (1997). Effective Bayesian inference for stochastic programs. In *National Conference on Artificial Intelligence (AAAI)*, pages 740–747.

Milch, B. and Russell, S. J. (2006). General-purpose MCMC inference over relational structures. In *Uncertainty in Artificial Intelligence(UAI)*.

Minka, T., Winn, J., Guiver, J., and Kannan, A. (2009). Infer.NET 2.3.

Pfeffer, A. (2007a). The design and implementation of IBAL: A general-purpose probabilistic language. In *Statistical Relational Learning*, pages 399–432.

Pfeffer, A. (2007b). A general importance sampling algorithm for probabilistic programs. Technical report, Harvard University TR-12-07.

Wingate, D., Goodman, N. D., Stuhlmüller, A., and Siskind, J. M. (2011). Nonstandard interpretations of probabilistic programs for efficient inference. In *Neural Information Processing Systems (NIPS)*, pages 1152–1160.