

可移植概率编程框架

The Framework for Portable Probabilistic Programming

李伟 朱其立

Wei Li Kenny Zhu

电子信息与电气工程学院

School of Electronic Information and Electrical Engineering

摘要

概率程序指的是在正常的函数式程序或者命令式程序的基础上多加了两个构造：（1）从一个给定的分布里采样一个随机值的能力，（2）基于程序已经观察到的变量特征，推断条件概率的能力。概率程序可以表示的模型覆盖多个应用领域，包括机器学习，数据检索，计算机视觉，编码理论，密码协议，和生物学等。

概率编程旨在减少表征概率模型的代码数量，降低开发时间，提供更丰富的模型，减少程序员在构建机器学习等相关领域的应用时所需的专业知识以降低进入这些领域的门槛，以及支持概率模型的整合。

概率编程使得概率推断的繁琐过程得以自动化进行。概率推断问题是指根据先验知识，由已知的变量信息推导出未知的变量信息的过程。所需要推断的结果会因应用场合需要而异。在概率编程中，构建模型与推导概率这两个过程被区分开来。

现在已经存在的概率编程语言或系统包括 BUGS, Church, FACTORIE, Infer.NET, Dimple 等。这些已有的系统都存在的问题是不利于跨平台的开发。因为程序开发人员们不得不适应、学习不同平台上的开发语言及开发库。

所以在这篇论文里，我们提出的是可移植概率编程框架，它能够被嵌入到多种常用的程序设计语言中，用户只需要学习这一种可移植概率编程语言，便可以在多个平台上开发概率程序。

我们设计了这一概率编程语言的语法 - 为旨在表征贝叶斯模型的声明式语言，并且实现了语法分析器。针对某几个给定的模型，用户输入的对于某一个概率的询问会由推断引擎自动得出，而不需要用户自己实现算法程序。推断引擎的实现是基于采样算法的近似推理。另外，我们利用一个简单包装与接口生成器 SWIG(Simplified Wrapper and Interface Generator) 实现了其他常用语言调用的接口。

关键词： 概率编程语言，概率图模型，概率推理，嵌入式程序设计语言

The Framework for Portable Probabilistic Programming

ABSTRACT

Probabilistic programs are usual functional or imperative programs with two additional constructs: (1) the ability to draw values at random from specified distributions, and (2) the ability to condition values of variables in a program based on observations. Probabilistic programs can represent models from diverse application areas including machine learning, information retrieval, computer vision, coding theory, cryptographic protocols, and biology.

Probabilistic programming aims to make the code of probabilistic models shorter, to reduce development time, to facilitate the construction of richer models, to require lower levels of expertise in building machine learning applications, and to support the construction of integrated models.

Probabilistic programming makes the inference process to be done automatically. Probabilistic inference is the problem of computing an explicit representation of the probability distribution implicitly specified by a probabilistic program. The desired output from inference may vary depending on the application. In Probabilistic programming, modeling and inference have been disentangled.

There are many existing probabilistic programming languages and systems including BUGS Lunn et al. (2000), Church Goodman et al. (2012), FACTORIE McCallum et al. (2009), Infer.NET Wang and Wand (2011), Dimple Hershey et al. (2012), etc. The problem is that it is not easy for those cross-platform developments where the programmers have to get accustomed to the different kinds of probabilistic programming

languages or libraries. Henceforth, in this paper, what we proposed is the Portable Probabilistic Programming Framework that can be embedded in every programming language that people commonly used. We designed the syntax for the portable probabilistic programming language which is a declarative programming language targeting to describe Bayesian networks. The design of the language is based on BUGS Lunn et al. (2000) and is more specific and efficient for describing the probabilistic models. The description of the models using the portable probabilistic programming language is separated from the code of the host language as well as the conditional query, which can enhance the reusability of the probabilistic models. The parser is implemented and the inference engine can calculate the result of the query automatically. The inference algorithm is based on sampling, which is efficient and lightweight to implement. Additionally, the APIs for other languages are attached leveraging the wrapper tool - SWIG(Simplified Wrapper and Interface Generator) Beazley *et al.* (1996).

Keywords: Probabilistic programming language, probabilistic graphical model, probabilistic inference, embedded programming language

Contents

1	Introduction	1
1.1	Probabilistic Programs	2
1.2	Probabilistic Graphical Model	4
1.2.1	Bayesian Networks	7
1.2.2	Markov Networks	8
1.3	Probabilistic Inference	9
1.3.1	Markov Chain Monte Carlo	10
1.4	Probabilistic Programming Language	12
2	Related Work	15
2.1	Probabilistic Languages	15
2.1.1	BUGS	16
2.1.2	Church	16
2.1.3	Infer.NET	16
2.1.4	FACTORIE	17
2.2	Lightweight Implementation of Probabilistic Programming Language .	17
2.3	Problem	18
3	Approach	19
3.1	Overview	19
3.2	Syntax of Programming Language	21
3.3	Probabilistic Distribution Library	25
3.4	Inference Engine	29
3.4.1	Rejection Sampling	30
3.4.2	Metropolis-Hastings Algorithm	31
3.4.3	From Samples to Probabilities	33
3.5	APIs for Other Languages	34
3.5.1	C API	36
3.5.2	Python API	37
3.5.3	Java API	38
3.5.4	Ocaml API	38
4	Implementation	42
4.1	Implementation	42
4.1.1	Parser	42
4.1.2	Probabilistic Distributions	44
4.1.3	Inference Engine	44
4.1.4	API	48
4.2	Case Study	48
5	Conclusion	50

Chapter 1 Introduction

Probabilities describe degrees of belief, and probabilistic inference describes rational reasoning under uncertainty. Undoubtedly, probabilistic models have exploded onto all the sciences of inference under uncertainty: machine learning, information retrieval, applied statistics and artificial intelligence. However, it's not easy for common programmers to describe the probabilistic models and encode the probabilistic inference process. Especially when the probabilistic models become complicated, the complexity to describe them and to perform inference will be raised subsequently. Just as programming beyond the simplest algorithms requires tools for abstraction and composition, complex probabilistic modeling requires new progress in model representation - probabilistic programming languages. Probabilistic programming languages are in the spotlight, which provide compositional means for describing complex probability distributions and performing efficient probabilistic inference. Goodman (2013).

The goal of probabilistic programming is to enable probabilistic modeling and machine learning to be accessible to the working programmer, who has sufficient domain expertise, but perhaps not enough expertise in probability theory or machine learning. We wish to hide the details of inference inside the compiler and runtime, and enable the programmers to express models using their domain expertise and dramatically increase the number of programmers who can benefit from probabilistic modeling. In probabilistic programming, modeling and inference have been disentangled.

In this chapter, the background information of probabilistic programming will be introduced. In section 1.1, we will illustrate what is probabilistic programs and will give some simple examples. In section 1.2, we will introduce probabilistic graphical models, especially the relationships between probabilistic programs and other probabilistic models that readers may have encountered before. Two kinds of probabilistic graphical models will be discussed: Bayesian networks and Markov networks. In section 1.3, we will elaborate on the probabilistic inference technologies and show some examples of probabilistic inference problems. In section 1.4, the formalized probabilistic pro-

programming language will be defined and the difference between common programming languages and probabilistic programming languages according to syntax and semantics will be illustrated.

1.1 Probabilistic Programs

Probabilistic programs are “usual” programs (written in languages like C, Java, Ocaml or Lisp) with two additional constructs: (1) the ability to draw values at random from distributions (like gaussian, gamma, etc.), and (2) the ability to condition values of variables in a program via observe statements (which can incorporate data from real world observations into a probabilistic program). Gordon et al. (2014). A variety of probabilistic programming languages and systems have been proposed, including BUGS Lunn et al. (2000), Church Goodman et al. (2012), FACTORIE McCallum et al. (2009), Infer.NET Wang and Wand (2011), Dimple Hershey et al. (2012), etc. However, unlike usual programs which are written for the purpose of being executed, the purpose of a probabilistic program is to implicitly specify a probability distribution.

We introduce the syntax and semantics of probabilistic programs using two simple probabilistic programs from Figure 1.1. This is an example written in **Probabilistic C**. Drawing from a Bernoulli distribution with mean 0.5 can be seen as tossing fair coins. The program in the top, Example 1(a), tosses two fair coins, and respectively assigns the outcomes of these coin tosses to the Boolean variables c_1 and c_2 , and returns (c_1, c_2) . The semantics of this program is the expectation of its return value. In this case, the return value will be $(1/2, 1/2)$. Since we have that

$$P(c_1 = false, c_2 = false) =$$

$$P(c_1 = false, c_2 = true) =$$

$$P(c_1 = true, c_2 = false) =$$

$$P(c_1 = true, c_2 = true) = 1/4$$

we have that the expectation on the return value is given by

$$1/4 \times (0, 0) + 1/4 \times (0, 1) + 1/4 \times (1, 0) + 1/4 \times (1, 1) = (1/2, 1/2)$$

where we treated true as 1 and false as 0.

The program in Example 1(b) is slightly different from Example 1(a). It has an observe statement $observe(c_1 \| c_2)$ before returning the value of (c_1, c_2) . The observe statement blocks runs which do not satisfy the boolean expression $c_1 \| c_2$ and does not permit those executions to happen. Executions that satisfy $c_1 \| c_2$ are permitted to happen. It is sort of a condition that the variables need to meet. Thus the semantics of the program is the expected return value conditioned by $(c_1 \| c_2)$, the permitted executions. Since conditioning by permitted executions yields

$$P(c_1 = false, c_2 = false) = 0$$

$$P(c_1 = false, c_2 = true) =$$

$$P(c_1 = true, c_2 = false) =$$

$$P(c_1 = true, c_2 = true) = 1/3$$

we have that the different expected return value which is

$$0 \times (0, 0) + 1/3 \times (0, 1) + 1/3 \times (1, 0) + 1/3 \times (1, 1) = (2/3, 2/3).$$

Another example of the probabilistic programs is for Latent Dirichlet Allocation (LDA), which can be seen in Figure 1.2. In the LDA example, the words in blue represent probability distributions with the arguments as parameters of the distributions. Each time a value can be sampled from the specified probability distributions, such as **dirichlet**, **multinomial**, **gaussian**, etc. The trace of the sampled value meet the requirement of the corresponding probability distribution. Compared with the LDA program coding in some common languages like C, Python, or Matlab in hundreds of lines, which contains both of the modeling code and inference code, programming in probabilistic programming language is a big relief for programmers and is much easier


```
1: bool c1, c2;  
2: c1 = Bernoulli(0.5);  
3: c2 = Bernoulli(0.5);  
4: return(c1, c2);
```

1(a)

```
1: bool c1, c2;  
2: c1 = Bernoulli(0.5);  
3: c2 = Bernoulli(0.5);  
4: observe(c1 || c2);  
5: return(c1, c2);
```

1(b)

Figure 1.1 Simple probabilistic programs

for the programmers to read the program.

1.2 Probabilistic Graphical Model

Probabilistic programs can be used to represent probabilistic graphical models Koller and Friedman (2009), which use graphs to denote conditional dependences between random variables.

Probabilistic graphical models(PGM) use a graph based representation as the basis for compactly encoding a complex distribution over a high-dimensional space. In the graphical representation illustrated in Figure 1.3, the nodes correspond to the variables in the domain, and the edges correspond to direct probabilistic interactions between them. For example, in Figure 1.3, the top of (a) illustrates one possible graph structure for a patient example, where `Flu` depends on `Season` and `Congestion` depends on `Hayfever`. In this graph, we can see that there is no direct edge between `Muscle Pain` and `Season`, but both interact directly with `Flu`. There are two perspectives that one can use to interpret the structure of this graph. From one perspective, the graph is a compact representation of a set of independencies that hold in the distribution; these properties take the form X is independent of Y given Z , denoted $(X \perp Y | Z)$, for some subsets of variables X, Y, Z . In this example, the distribution satisfies the conditional

```

function X = lda()

K = 10;
for k=1:K
    topics(k,:) = dirichlet( 1, vocab_size );
end;

for d=1:num_docs
    topic_dist = dirichlet( 1, K );
    for w=1:num_words(d)
        topic = multinomial( topic_dist );
        X{d}(w) = multinomial( topics(topic,:) );
    end;
end;

return;

```

Figure 1.2 Probabilistic program example: LDA.

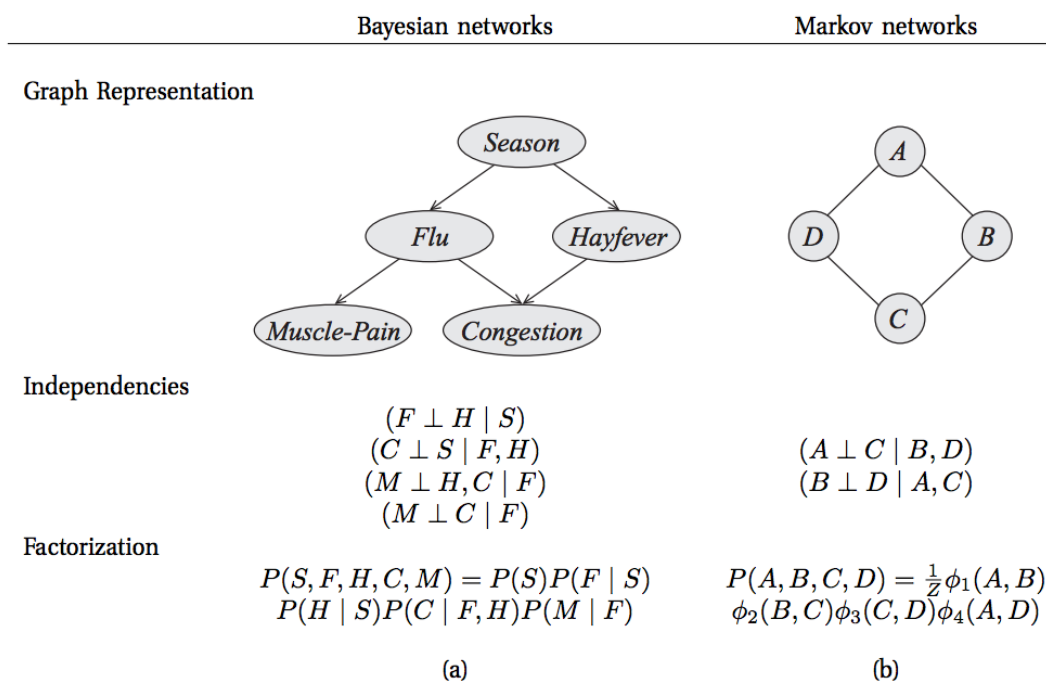


Figure 1.3 Probabilistic graphical models in different perspectives: the graphical representation (top); the independencies induced by the graph structure (middle); the factorization induced by the graph structure (bottom). (a) A sample Bayesian network. (b) A sample Markov network.

independence ($\text{Congestion} \perp \text{Season} | \text{Flu}, \text{Hayfever}$). This statement asserts that

$$P(\text{Congestion} | \text{Flu}, \text{Hayfever}, \text{Season}) = P(\text{Congestion} | \text{Flu}, \text{Hayfever});$$

that is, if we are interested in the distribution over the patient having congestion, and we know whether he has the flu and whether he has hayfever, the season is no longer informative. But this assertion does not imply that `Season` is independent of `Congestion`. Figure 1.3a (middle) shows the set of independence assumptions associated with the graph in Figure 1.3a (top).

The other perspective is that the graph model describes a skeleton for compactly representing a high dimensional factor graph. Rather than encoding the probability of every possible assignment to factor all of the variables in our domain, we can “break up” the distribution into smaller factors, each over a much smaller space of possibilities. We can then define the overall joint distribution as a product of these factors. For example, Figure 1.3a (bottom) shows the factorization of the distribution associated with the graph in Figure 1.3a (top). It asserts, for example, that the probability of the event spring, no flu, hayfever, sinus congestion, muscle pain can be obtained by multiplying five numbers:

$$P(\text{Season} = \text{spring})$$

$$P(\text{Flu} = \text{false} | \text{Season} = \text{spring})$$

$$P(\text{Hayfever} = \text{true} | \text{Season} = \text{spring})$$

$$P(\text{Congestion} = \text{true} | \text{Hayfever} = \text{true}, \text{Flu} = \text{false})$$

$$P(\text{MusclePain} = \text{true} | \text{Flu} = \text{false})$$

The graph structure defines the factorization of a distribution P associated with it - the set of factors and the variables that they encompass.

We will describe two families of graphical representations of distributions. One, called Bayesian networks, uses a directed graph, as shown in Figure 1.3a (top). The second, called Markov networks, uses an undirected graph, as illustrated in Figure 1.3b

(top). It can be viewed as defining a set of independence assertions (Figure 1.3b (middle)) or as encoding a compact factorization of the distribution as showed in Figure 1.3b (bottom). Both representations provide the duality of independencies and factorization, but they differ in the set of independencies they can encode and in the factorization of the distribution that they induce. In this paper we will focus more on Bayesian networks.

1.2.1 Bayesian Networks

A Bayesian network structure is a directed, acyclic graph G , where each vertex s of G is interpreted as a random variable X_s (with unspecified distribution) Heckerman et al. (1995). A Bayesian network (G, P) consists of

- a BN structure G and
- a set of conditional probability distributions (CPDs) $P(X_s | Pa_{X_s})$, where Pa_{X_s} are the parents of node X_s such that
- (G, P) defines a joint distribution

$$P(X_1, \dots, X_n) = \prod_i P(X_i | Pa_{X_i})$$

For Bayesian networks, there are two kinds of different inference problems. Heckerman (1998).

Learning Probabilities in a Bayesian Network

Bayesian networks can be used to answer probabilistic queries given the prior probabilities and conditional probabilities. For instance, in a Bayesian network there are two random variable X, Y where X is observed and Y is unobserved. And we know that the dependency probability of $P(Y|X)$, then we can learn the joint probability of X, Y

where

$$P(X, Y) = P(X) \times P(Y|X).$$

The process of computing the posterior distribution of variables given evidence is called *probabilistic inference*. More details can be seen in Section 1.3. On the other hand, we can choose a value for a random variable in its domain to maximize a likelihood probability or minimize a cost function in a Bayesian network.

Learning parameters and structures

Sometimes the given Bayesian network has some unobserved variables without knowing the exact distributions that the variables follow. Then we can estimate the parameters of the distribution based on the probability of observed variables and the structure of the graph. We can also estimate the parameter from data, for example, using the maximum likelihood approach. There are also some other approaches to this problem like the expectation maximization(EM) algorithm and the Viterbi algorithm for Hidden Markov Model(HMM). There are also some sampling algorithms that to learn the parameters by viewing the parameters as additional hidden variables.

For structure learning, when defining the probabilistic graphical model is pretty complex for humans or there are many probabilistic models one can choose from, we can learn the network structure from data. Doing structure learning automatically is still a challenge.

1.2.2 Markov Networks

Markov network, or *Markov random field* is a set of random variables having a Markov property described by an undirected graph. A Markov random field is similar to a Bayesian network in its representation of dependencies. The differences being that Bayesian networks are directed and acyclic, whereas Markov networks are undirected and may be cyclic. Thus, a Markov network can represent certain dependencies that a Bayesian network cannot (such as cyclic dependencies). On the other hand, it can't

represent certain dependencies that a Bayesian network can (such as induced dependencies). Kindermann et al. (1980).

In our framework, the language targets to describe Bayesian networks and solves the query of inferring unobserved variables by probabilistic inference.

1.3 Probabilistic Inference

Probabilistic inference is the problem of computing an explicit representation of the probability distribution implicitly specified by a probabilistic program. If the probability distribution is over a large number of variables, an explicit representation of the joint probability distribution may be both difficult to obtain efficiently, and unnecessary in the context of specific application contexts. The desired output of inference may vary from different applications. For example, we may want to compute the expected value of some function f with respect to the distribution (which may be more efficient to calculate without representing the entire joint distribution). Alternatively, we may want to calculate the most likely value of the variables, which is the mode of the distribution. Or we may want to simply draw a set of samples from the distribution, to test some other system which expects inputs to follow the modeled distribution. Gordon et al. (2014).

Figure 1.4. gives an example of inference problem. The variable *Difficulty* describes the difficulty of the courses with probability 0.6 to be not hard, namely d_0 , and with probability 0.4 to be hard, that is d_1 . Other variables share the similar meaning with *Difficulty*. If we want to know the probability of the student getting *Letter* = l_1 under the condition that his grade *Grade* = g_1 , then this is an inference problem with the formal representation $P(L = l_1 | G = g_1)$. Also, if we want to know the probability of the student getting *Grade* = g_1 under he/she has already get the *Letter* = l_1 , which is a backward process compared with the previous conditional probability, this is also the problem of probabilistic inference with the formal representation $P(G = g_1 | L = l_1)$.

There are two kinds of inference technologies, exact inference and approximate inference. The exact inference methods include elimination algorithm, namely sum-

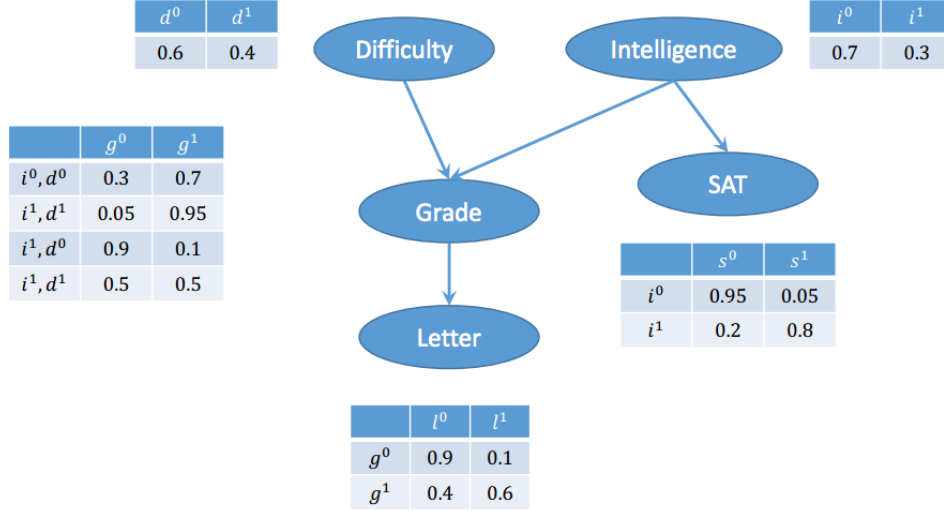


Figure 1.4 A student example to show probabilistic inference.

product algorithm, which computes the probability by summing the joint probability thus to eliminate the non-observed and non-query variables; message-passing algorithm, which is an improvement of elimination algorithm if we want to calculate more than one marginal probability. In message-passing algorithm, the intermediate terms of probability will be viewed as “messages” and the inference process will be considered as local computation and routing of messages. All of the exact inference algorithms have the complexity that is exponential to the size of the graph. The approximate inference algorithms include stochastic Markov chain Monte Carlo (MCMC) simulation Andrieu et al. (2003) which contains many sampling algorithms like Gibbs sampling, Metropolis-Hastings Algorithms and importance sampling, mini-bucket elimination, generalized belief propagation, and variational methods.

1.3.1 Markov Chain Monte Carlo

In this paper, we focused on the MCMC simulation methods to do inference. MCMC techniques are often applied to solve integration and optimisation problems in large dimensional spaces. In Bayesian inference and learning, given some unknown variables $x \in \mathcal{X}$ and data $y \in \mathcal{Y}$. the following typically intractable integration problems are

central to Bayesian statistics.

- Normalization. To obtain the posterior $p(x|y)$ given the prior $p(x)$ and likelihood $p(y|x)$, the normalising factor in Bayes' theorem needs to be computed

$$p(x|y) = \frac{p(y|x)p(x)}{\int_{\mathcal{X}} p(y|x')p(x')dx'}$$

- Marginalisation. Given the joint posterior of $(x, z) \in \mathcal{X} \times \mathcal{Z}$, we may often be interested in the marginal posterior

$$p(x|y) = \int_{\mathcal{Z}} p(x, z|y)dz.$$

- Expectation. The objective of the analysis is often to obtain summary statistics of the form

$$\mathbb{E}_{p(x|y)}(f(x)) = \int_{\mathcal{X}} f(x)p(x|y)dx$$

for some function of interest $f : \mathcal{X} \rightarrow \mathbb{R}^{n_f}$ integrable with respect to $p(x|y)$. Examples of appropriate functions include the conditional mean, in which case $f(x) = x$, or the conditional covariance of x where $f(x) = xx' - \mathbb{E}_{p(x|y)}(x)\mathbb{E}'_{p(x|y)}(x)$.

The idea of Monte Carlo simulation is to draw an i.i.d. set of samples $\{x^{(i)}\}_{i=1}^N$ from a target density $p(x)$ defined on a high-dimensional space \mathcal{X} . These N samples can be used to approximate the target density with the following empirical point-mass function

$$p_N(x) = \frac{1}{N} \sum_{i=1}^N \sigma_{x^{(i)}}(x)$$

where $\sigma_{x^{(i)}}(x)$ denotes the delta-Dirac mass located at $x^{(i)}$.

MCMC is a strategy for generating samples $x^{(i)}$ while exploring the state space \mathcal{X} using a Markov chain mechanism. To introduce Markov chains on finite state spaces, suppose that $x^{(i)}$ can only take s discrete values $x^{(i)} \in \mathcal{X} = x_1, x_2, \dots, x_s$. The

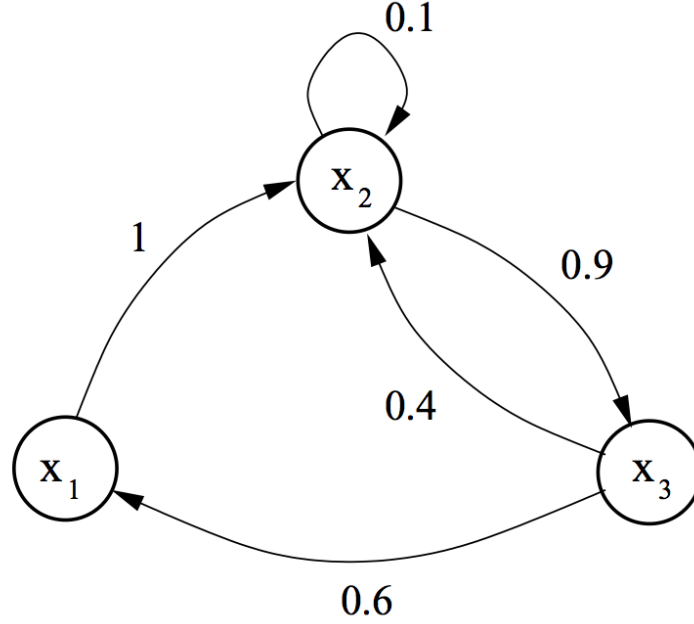


Figure 1.5 Transition graph for a Markov chain example

stochastic process $x^{(i)}$ is called a Markov chain if

$$p(x^{(i)} | x^{(i-1)}, \dots, x^{(1)}) = T(x^{(i)} | x^{(i-1)})$$

where T is a transition function. That is, the evolution of the chain in a space \mathcal{X} depends solely on the current state of the chain and fixed transition matrix. Figure 1.5 gives an example of transition graph for Markov chain with $\mathcal{X} = \{x_1, x_2, x_3\}$.

We leveraged two methods of MCMC technologies for approximate inference including rejection sampling Neal (2003) and Metropolis-Hastings algorithm Chib and Greenberg (1995). More details can be found in Chapter 3.

1.4 Probabilistic Programming Language

Probabilistic Programming Language (PPL) extends a well-specified deterministic programming language with primitive constructs for random choice. Just as with deterministic programming languages, there are probabilistic languages in the imperative, functional, and logical paradigms. We will take one of the probabilistic programming language **Probabilistic C** Paige and Wood (2014) as example, which is a C-like imper-

x	\in	Vars	
uop	$::=$	\dots	C unary operators
bop	$::=$	\dots	C binary operators
φ, ψ	$::=$	\dots	logical formula
\mathcal{E}	$::=$	x c $\mathcal{E}_1 \text{ bop } \mathcal{E}_2$ $\text{uop } \mathcal{E}$	expressions variable constant binary operation unary operation
\mathcal{S}	$::=$	$x = \mathcal{E}$ $x \sim \text{Dist}(\bar{\theta})$ skip observe (φ) $\mathcal{S}_1; \mathcal{S}_2$ if \mathcal{E} then \mathcal{S}_1 else \mathcal{S}_2 while \mathcal{E} do \mathcal{S}	statements deterministic assignment probabilistic assignment skip observe sequential composition conditional composition while-do loop
\mathcal{P}	$::=$	$\mathcal{S} \text{ return } (\mathcal{E})$	program

Figure 1.6 Syntax of PROB

ative programming language with two additional statements:

1. The probabilistic assignment “ $x \sim \text{Dist}(\bar{\theta})$ ” draws a sample from a distribution *Dist* with a vector of parameters $\bar{\theta}$, and assigns it to the variable x . For instance, the statement “ $x \sim \text{Gaussian}(\mu, \sigma)$ ” draws a value from a *Gaussian* distribution with mean μ and standard deviation σ , and assigns it to the variable x .
2. The observe statement “*observe*(φ)” conditions a distribution with respect to a predicate or condition φ that is defined over the variables in the program. In particular, every valid execution of the program must satisfy all conditions in observe statements that occur along the execution.

The syntax of **Probabilistic C** is formally described in Figure 1.6. A program consists of a statement and a return expression. Variables have base types such as int, bool, float and double. Expressions include variables, constants, binary and unary operations.

Statements include primitive statements (deterministic assignment, probabilistic assignment, observe, skip) and composite statements (sequential composition, conditionals and loops).

In summary, probabilistic graphical models are the representations of the structure of probability distributions. One can derive the explicit representation of probability distribution by performing probabilistic inference. Probabilistic programming language can describe probabilistic graphical models and query some conditional probability or joint probability. So probabilistic programs have two extra feature than usual programs:

1. Draw samples from probability distribution
2. Query some probability automatically.

In Chapter 2, the related work will be introduced including the state of the art probabilistic programming languages and systems. In Chapter 3, we will elaborate on the approach of the design of the portable probabilistic programming framework. More specifically, an overview of our framework will be showed in Section 3.1. The syntax of the declarative language is illustrated in Section 3.2 and the implementation of the probabilistic library is showed in Section 3.3. The algorithms of probabilistic inference are elaborated in Section 3.4. And the principles and usage of the APIs for other common programming languages is illustrated in Section 3.5. In Chapter 4, the implementations of our framework and the case studies of the performance of the framework will be showed. In Chapter 5, we will conclude our work and the future work that can be done based on our framework.

Chapter 2 Related Work

This area of research is becoming more and more popular in recent years with more and more publications in top conferences such as POPL and NIPS, attracting researchers from universities such as MIT and Stanford and industry research institutions such as Microsoft Research. However, to our best knowledge, this area has not even been studied in China.

In this chapter, we will introduce the state of the art probabilistic programming languages and systems, algorithms of inference engine, as well as the current problems existed for probabilistic programming.

2.1 Probabilistic Languages

We have evaluated the existing probabilistic programming systems including BUGS, Church, FACTORIE, Infer.NET, Dimple, etc. More specifically, BUGS Lunn et al. (2000) is a language for specifying finite graphical models and accompanying software for performing Bayesian inference Using Gibbs Sampling. Church Goodman et al. (2012) is a universal probabilistic programming language, extending Scheme with probabilistic semantics, and is well suited for describing infinite-dimensional stochastic processes and other recursively-defined generative processes. Factorie McCallum et al. (2009) is a Scala library for creating relational factor graphs, estimating parameters and performing inference. Infer.NET Wang and Wand (2011) is a software library developed by Microsoft for expressing graphical models and implementing Bayesian inference using a variety of algorithms within the .NET platform. Dimple Hershey et al. (2012) is a software tool that performs inference and learning on probabilistic graphical models via belief propagation algorithms or sampling based algorithms. IBAL Pfeffer (2001) is a rational programming language for probabilistic and decision-theoretic agents. It integrates Bayesian parameter estimation and decision-theoretic utility maximization thoroughly into the framework. But it only works with discrete data types.

Alchemy Poon and Domingos (2008) represents Markov logic networks, and undirected graphical model over log-linear-weighted first order logic, which is inherently discrete. In summary, all these probabilistic programming languages can only be used in specific environment and inherent some features from the domain languages thus can not be easily used in cross-platform developments.

2.1.1 BUGS

The BUGS (Bayesian inference Using Gibbs Sampling) project is concerned with flexible software for the Bayesian analysis of complex statistical models using Markov chain Monte Carlo(MCMC) methods. Lunn et al. (2000). BUGS allow users to specify a generative model imperatively, then it does inference with a Gibbs sampler, thus being able to handle a wide variety of different sorts of models. BUGS can also be used from R. Actually, the model definition language in BUGS itself is R-like but not actually R. BUGS is not a Turing-complete language.

2.1.2 Church

Church extends (the purely functional subset of) Scheme with elementary random primitives(ERP), such as flip (a bernoulli), multinomial, and gaussian. In addition, Church includes language constructs that simplify modeling. For instance, `mem`, a higher-order procedure that memorizes its input function, is useful for describing persistent random properties and lazy model construction. (see Goodman (2013), Goodman et al. (2012)).

2.1.3 Infer.NET

Infer.NET focuses on message-passing inference Bishop *et al.* (2006), written in C#. It has a visualized representation of the declared probabilistic graphical models to help the users to check correctness. It offers several options of inference algorithms including both exact inference and approximate inference. Users can only use Infer.NET in the .NET framework.

2.1.4 FACTORIE

Factorie McCallum et al. (2009) focuses on factor graphs and discriminative undirected models. They implemented a library for Scala to allow sampling from probabilistic distributions and inference automatically. *Factorie* has good scalability over large parameters in the probabilistic graphical models. Different from previous probabilistic programming languages, it targets Markov networks rather than Bayesian networks.

2.2 Lightweight Implementation of Probabilistic Programming Language

Wingate et al. (2011) proposed a general method for transforming arbitrary programming languages into probabilistic programming languages with an accompanying implementation of straightforward Markov chain Monte Carlo inference engines. Their main contribution lies in the naming strategy where they give each random choice of a fixed program a unique “name” depending on its position in a given execution trace. Such that the stochastic functions can be converted into deterministic ones as the names can be used to look up the return value in a database that stores the values of all the random choices in a fixed program. Wingate et al. (2011) showed how *nonstandard interpretations* of probabilistic programs can be used to perform efficient inference algorithms. In their method, information about the structure of the distributions (which is the dependencies or gradients in the probabilistic graphical models) is derived as monad-like side computation at the same time of executing the program. Meanwhile, the interpretations can be coded easily with some special-purpose objects and operator overloading. They promoted the inference efficiency performance by using the structure information of distribution as part of the variety of inference algorithms.

Additionally, because the program is in a machine-readable form, various of techniques from compiler design and program analysis can be used in the inference engine. Gordon et al. (2013) designed a new model-learner pattern for Bayesian reasoning. In their work, a new probabilistic programming abstraction was proposed, a typed Bayesian model, based on a pair of probabilistic expressions for the prior and sampling

distributions. Also, Claret et al. (2013) presented a new algorithm for Bayesian inference over probabilistic programs, which is based on data flow analysis techniques from the program analysis community.

2.3 Problem

Although currently there are many probabilistic programming languages and systems, as stated above, the problem is that it is not easy for those cross-platform developments where the programmers have to get accustomed to the different kinds of probabilistic programming languages or libraries. Henceforth, we proposed the Portable Probabilistic Programming Framework that can be embedded in every programming language people commonly used.

Chapter 3 Approach

3.1 Overview

The structure of the framework is showed in Figure 3.1. Users can define probabilistic models in our model declaring language, where they can directly call the functions of distributions without implementing the distributions. They can load the models and write queries in their domain languages with the APIs. Then the inference engine will infer the queries automatically based on the sampling traces. Thus the desired probability will be derived.

There are four steps in the design and implementation of the portable probabilistic programming framework. At first we designed the syntax of the embedded programming language, which targets the description of the Bayesian networks. The design of the language is based on BUGS, but the syntax is more specific and efficient to be lightweight for the portable characteristic. Also the syntax can be expressive for most of the probabilistic models. The description of the models using the portable probabilistic programming language is separated from the code of the host language as well of the conditional query, which can enhance the reusability of the probabilistic models. The parser is implemented and the inference engine is generated automatically based on the conditional query. More details will be illustrated in Section 3.2.

We implemented the probabilistic library for most of the probability distribution such as Gaussian, Gama, Beta, etc. Our probabilistic programs define distributions by defining a distribution over possible execution traces. The distribution is fully specified by a generative procedure. More details can be found in Section 3.3.

The inference algorithm is based on the MCMC sampling, including rejection sampling and Metropolis-Hastings Algorithm, which is efficient and lightweight to implement. We will elaborate more on the mechanism of the inference engine in Section 3.4.

Additionally, we implement the APIs for other languages leveraging the existing

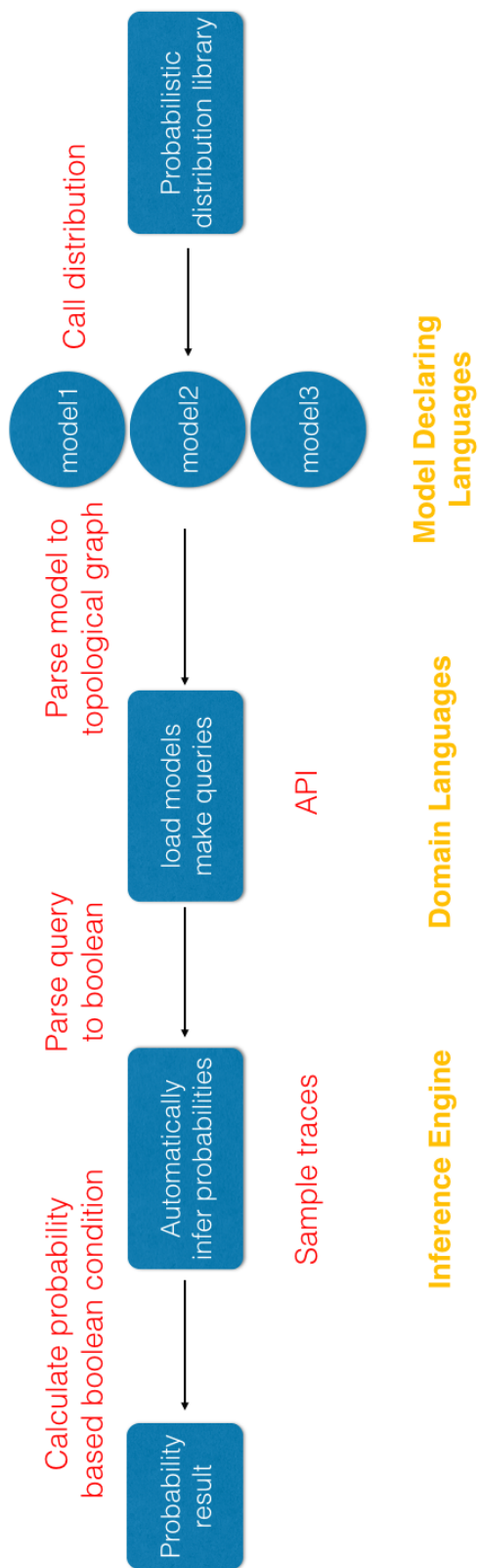


Figure 3.1 Overview of the Portable Probabilistic Programming Framework.

```
model flip_example() {  
  public flip  
  public x  
  private x1  
  private x2  
  
  flip ~ dbern(0.5)  
  x1 ~ dnorm(0, 1)  
  x2 ~ dgamma(1, 1)  
  x = if flip then x1 else x2 + 2  
}
```

Figure 3.2 Flip example, describing the probabilistic model in our language.

development tool: SWIG (Simplified Wrapper and Interface Generator). More details for the APIs are in Section 3.5.

Our main contribution lies in the design of the portable probabilistic programming language to make it portable, the implementation of the probability distribution library and the lightweight implementation of the inference engine.

3.2 Syntax of Programming Language

We will give some intuition of our portable probabilistic programming language by giving an example declaring the flip model, which is showed in Figure 3.2.

The flip example describes a probabilistic model that has the distribution over variables as showed in Figure 3.3. The grammar for this specific example is showed in Figure 3.4. Under this syntax, the parser will parse the probabilistic program and generate the Bayesian network as the user described. The inference is done based on the probabilistic graph. The Bayesian network of the flip example is showed in Figure 3.5.

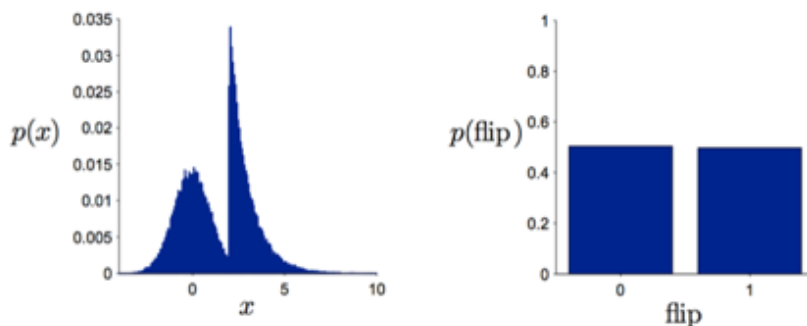


Figure 3.3 Flip example, the implied distributions over variables.

```

model -> "model" name "(" ")" "{" decls stmts "}"

decls -> decl
      | decl decls

stmts -> stmt
      | stmt stmts

decl -> "public" name
      | "private" name

stmt -> name "~" name "(" expr_seq ")"
      | name "=" expr

expr_seq -> expr
          | expr "," expr_seq

expr -> "if" expr "then" expr "else" expr
      | add_expr

add_expr -> primary
          | primary "+" add_expr

primary -> numerical_value
         | name

numerical_value -> integer_value
                 | integer_value "." integer_value

integer_value -> [0-9]+

name -> [a-zA-Z_][0-9a-zA-Z_]*
  
```

Figure 3.4 The grammar for the flip example.

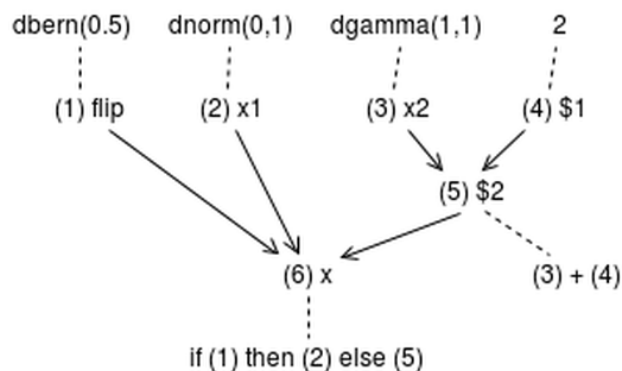


Figure 3.5 The generated Bayesian network for the flip example.

The grammar for the portable probabilistic programming language is as below:

$$\begin{aligned}
 models &\rightarrow model \\
 &\quad | \quad model \ models \\
 model &\rightarrow \textcolor{red}{“model”} \ name \ \textcolor{red}{“(”} \quad \quad \quad \textcolor{red}{“)”} \ \textcolor{red}{“\{”} \ decls \ stmts \ \textcolor{red}{“\}”} \\
 &\quad | \quad \textcolor{red}{“model”} \ name \ \textcolor{red}{“(”} \ model_params \ \textcolor{red}{“)”} \ \textcolor{red}{“\{”} \ decls \ stmts \ \textcolor{red}{“\}”} \\
 model_params &\rightarrow name \\
 &\quad | \quad name \ \textcolor{red}{“,”} \ model_params \\
 decls &\rightarrow decl \\
 &\quad | \quad decl \ decls \\
 stmts &\rightarrow stmt \\
 &\quad | \quad stmt \ stmts \\
 decl &\rightarrow \textcolor{red}{“public”} \ variable \\
 &\quad | \quad \textcolor{red}{“private”} \ variable \\
 stmt &\rightarrow variable \ \textcolor{red}{“\sim”} \ name \ \textcolor{red}{“(”} \ expr_seq \ \textcolor{red}{“)”} \\
 &\quad | \quad variable \ \textcolor{red}{“=”} \ expr \\
 &\quad | \quad \textcolor{red}{“for”} \ name \ \textcolor{red}{“=”} \ expr \ \textcolor{red}{“to”} \ expr \ \textcolor{red}{“\{”} \ stmts \ \textcolor{red}{“\}”} \\
 expr_seq &\rightarrow expr \\
 &\quad | \quad expr \ \textcolor{red}{“,”} \ expr_seq
 \end{aligned}$$

$$\begin{aligned}
 \text{expr} &\rightarrow \text{"if"} \text{ expr } \text{"then"} \text{ expr } \text{"else"} \text{ expr} \\
 &\quad | \text{"new"} \text{ name } \text{"(" } \text{")"} \\
 &\quad | \text{add_expr} \\
 \text{add_expr} &\rightarrow \text{term} \\
 &\quad | \text{term } \text{"+"} \text{ add_expr} \\
 &\quad | \text{term } \text{"-"} \text{ add_expr term} \rightarrow \text{primary} \\
 &\quad | \text{primary } \text{"*"} \text{ term} \\
 &\quad | \text{primary } \text{" /"} \text{ term} \\
 \text{primary} &\rightarrow \text{numerical_value} \\
 &\quad | \text{variable} \\
 &\quad | \text{function_call} \\
 &\quad | \text{"(" } \text{expr } \text{")"} \\
 &\quad | \text{"-"} \text{ primary} \\
 \text{function_call} &\rightarrow \text{name } \text{"(" } \text{expr_seq } \text{")"} \\
 \text{variable} &\rightarrow \text{name} \\
 &\quad | \text{field_var} \\
 &\quad | \text{index_var} \\
 \text{field_var} &\rightarrow \text{name } \text{"."} \text{ name} \\
 \text{index_var} &\rightarrow \text{name } \text{"[" } \text{expr_seq } \text{"]"} \\
 \text{numerical_value} &\rightarrow \text{integer_value} \\
 &\quad | \text{integer_value } \text{"."} \text{ integer_value} \\
 \text{integer_value} &\rightarrow [0 - 9]^+ \\
 \text{name} &\rightarrow [a - z A - Z_][0 - 9 a - z A - Z_]*
 \end{aligned}$$

Users can declare many probabilistic graphical models in our declarative program-

```

1  model hidden_markov_model(n, a, b) {
2      // n -- chain length
3      // a -- number of possible states
4      // b -- number of possible observations
5      public states[n]    // hidden states
6      public observ[n]    // observations
7      public trans[a,a]   // transition matrix of size a*a
8      public emiss[a,b]   // emission matrix of size a*b
9      public start[a]     // start probabilities of size a
10
11      states[0] ~ dcat(start)
12      for i = 1 to n-1 {
13          states[i] ~ dcat(trans[states[i-1]])
14      }
15
16      for i = 0 to n-1 {
17          observ[i] ~ dcat(emiss[states[i]])
18      }
19  }
20

```

Figure 3.6 An HMM example

ming language. Figure 3.6 shows an example of Hidden Markov Model(HMM) described in our language and Figure 3.7 shows an example of Latent Dirichlet Allocation(LDA).

In our design, the declarations of the Bayesian networks and the query of conditional probability is separated. Users can describe the model in a model file and load the model in their domain languages. In this way, users can reuse their models and load several probabilistic graphical models at the same time. Also their queries can base on these models which have a better structure than mixing the models up. Examples of writing the queries and load models can be found in 3.5.

3.3 Probabilistic Distribution Library

We implemented the probabilistic library for most of the probability distribution such as Gaussian, Gama, Beta, etc. Our probabilistic programs define distributions by defining a distribution over possible execution traces. The distribution is fully specified by a generative procedure. Some complex distributions are crafted compositionally. An

```

1  model latent_dirichlet_allocation(k, num_docs, num_words[],
2  vocab_size) {
3      for i = 1 to k {
4          topics(i, :) = dirichlet(1, vocab_size);
5      }
6
7      for i = 1 to num_docs {
8          topic_dist = dirichlet(1, k);
9          for j = 1 to num_words(i) {
10             topic = multinomial(topic_dist);
11             X{i}(j) = multinomial(topics(topic, :));
12         }
13     }
14 }

```

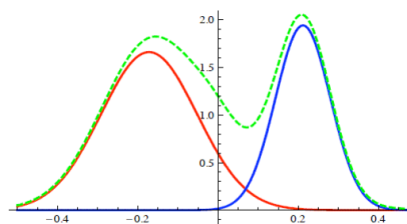
Figure 3.7 An LDA example

```

function X = gmm()
    if ( rand > 0.5 )
        X = -0.2 + randn
    else
        X = 0.2 + 0.5*randn
    end;
return;

>> gmm
-0.21
>> gmm
0.3

```



Complex distributions are crafted
compositionally

Figure 3.8 Implementation of GMM distribution example.

example is showed on how to implement a Gaussian Mixture Model distribution in Figure 3.8. As can be seen in Figure 3.8., everytime we run the function *gmm()*, it will return a value based on the distribution implementation. And the trace of the returned value will meet the requirements of the density probability the program has defined.

Most of the distributions in our library are implemented based on Normal distribution and Bernoulli distribution according to mathematical algorithms. The probability distributions in our framework contain:

Code	Distribution
dbern	Bernoulli distribution
dnorm	Normal distribution
dgamma	Gamma distribution
dbinomial	Binomial distribution
dmultinomial	Multinomial distribution
ddirich	Dirichlet distribution
dbeta	Beta distribution
duniform	Uniform distribution
duniform_discrete	Discrete uniform distribution
dpoisson	Poisson distribution
dcat	Categorical distribution

There are many algorithms to generate values from some specific probability distribution. For example, to generate values from Gaussian distribution, there is Box-Muller transform method Box et al. (1958) and Ziggurat algorithm Marsaglia and Tsang (2000). We leveraged Ziggurat algorithm in our framework since it is faster and still exact.

As is showed in Figure 3.9, each rectangular is a layer. Given uniform random variables $U_0, U_1 \in [0, 1)$, the ziggurat algorithm can be described as:

1. Choose a random layer $0 \leq i < n$.
2. Let $x = U_0 \times x_i$
3. If $x < x_i + 1$, return x .
4. If $i == 0$, generate a point from the tail using the fallback algorithm.
5. Let $y = y_i + U_1 \times (y_i + 1 - y_i)$.
6. Compute $f(x)$. If $y < f(x)$, return x .
7. Otherwise, choose new random numbers and go back to step 1.

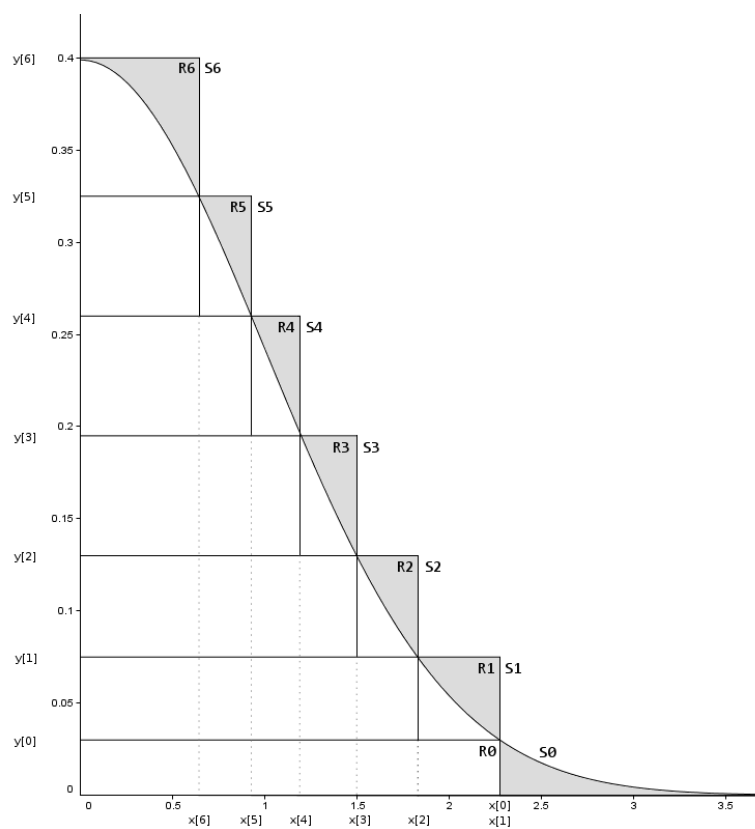


Figure 3.9 The ziggurat method with 7 rectangles.

3.4 Inference Engine

Calculating the distribution specified by a probabilistic program is called *probabilistic inference*. The inferred probability distribution is called the *posterior probability distribution*, and the initial guess made by the program is called the *prior probability distribution*. Let $(E; F)$ be a partitioning of the node indices of a graphical model into disjoint subsets, such that (X_E, X_F) is a partitioning of the random variables. There are two kinds of inference problems that we want to solve:

- Marginal probabilities:

$$p(x_E) = \sum_{x_F} p(x_E, x_F).$$

- Maximum a posteriori(MAP) probabilities:

$$p^*(x_E) = \max_{x_F} p(x_E, x_F).$$

From these basic computations we can obtain other quantities of interest. In particular, the *conditional probability* $p(x_E|x_F)$ is equal to

$$p(x_E|x_F) = \frac{p(x_E, x_F)}{\sum_{x_E} p(x_E, x_F)}.$$

The mechanism of our inference engine is based on sampling. Currently we used Rejection Sampling algorithm in the inference engine, which is straightforward and easy to implement. However, rejection Sampling is not as efficient as other sampling algorithms like Gibbs Sampling and Metropolis-Hastings algorithm. Henceforth, we added another optional inference algorithm in our inference engine: Metropolis-Hastings algorithm.

There are three problems we need to consider in stochastic sampling:

- how to generate samples
- how to incorporate observations

```

Set  $i = 1$ 
Repeat until  $i = N$ 
  1. Sample  $x^{(i)} \sim q(x)$  and  $u \sim \mathcal{U}(0,1)$ .
  2. If  $u < \frac{p(x^{(i)})}{Mq(x^{(i)})}$  then accept  $x^{(i)}$  and increment the counter  $i$  by 1. Otherwise, reject.
  
```

Figure 3.10 Rejection sampling algorithm. Here, $u \sim \mathcal{U}(0,1)$ denotes the operation of sampling a uniform random variable on the interval $(0,1)$.

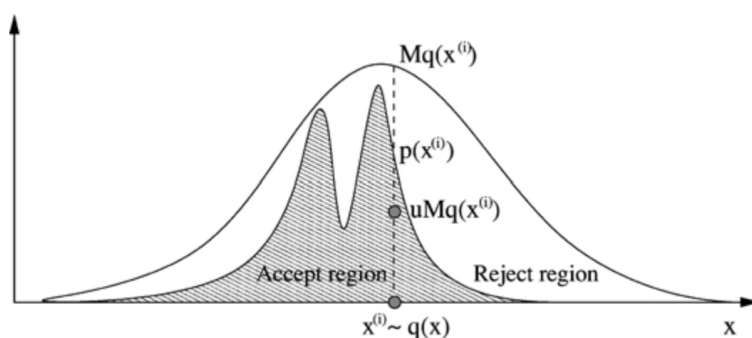


Figure 3.11 Rejection sampling: Sample a candidate $x^{(i)}$ and a uniform variable u . Accept the candidate sample if $uMq(x^{(i)}) < p(x^{(i)})$, otherwise reject it.

- how to infer probabilities from samples

The intuitive idea is to sample the traces as the program runs. If the trace meets the Boolean requirements of the condition, then record the trace, otherwise the trace is discarded. Then we calculate the number of the traces that can meet the Boolean requirement of the probability we want to get. Then calculating this number over the whole number of traces we recorded can derive the final conditional probability.

3.4.1 Rejection Sampling

Rejection sampling is a basic technique used to generate observations from a distribution. It is a type of Monte Carlo method. The method works for any distribution in \mathbb{R}^m with a density.

Rejection sampling is based on the observation that to sample a random variable one can sample uniformly from the region under the graph of its density function. We

```

1. Initialise  $x^{(0)}$ .
2. For  $i = 0$  to  $N - 1$ 
    - Sample  $u \sim \mathcal{U}_{[0,1]}$ .
    - Sample  $x^* \sim q(x^*|x^{(i)})$ .
    - If  $u < \mathcal{A}(x^{(i)}, x^*) = \min\left\{1, \frac{p(x^*)q(x^{(i)}|x^*)}{p(x^{(i)})q(x^*|x^{(i)})}\right\}$ 
         $x^{(i+1)} = x^*$ 
    else
         $x^{(i+1)} = x^{(i)}$ 

```

Figure 3.12 Metropolis-Hastings algorithm.

can sample from a distribution $p(x)$, which is known up to a proportionality constant, by sampling from another easy-to-sample proposal distribution $q(x)$ that satisfies $p(x) \leq Mq(x)$, $M < \infty$, using the accept/reject procedure describe in Figure 3.10 (see also Figure 3.11). Andrieu et al. (2003) The accepted $x(i)$ can be easily shown to be sampled with probability $p(x)$ Casella and Robert (1999). This simple method suffers from severe limitations. It is not always possible to bound $p(x)q(x)$ with a reasonable constant M over the whole space X . If M is too large, the acceptance probability

$$Pr(x \text{ accepted}) = Pr(u < \frac{p(x)}{Mq(x)}) = \frac{1}{M}$$

will be too small. This makes the method impractical in high-dimensional scenarios.

3.4.2 Metropolis-Hastings Algorithm

The *Metropolis-Hastings (MH) algorithm* is the most popular MCMC method. An MH step of invariant distribution $p(x)$ and proposal distribution $q(x^*|x)$ involves sampling a candidate value x^* given the current value x according to $q(x^*|x)$. The Markov chain then moves towards x^* with acceptance probability

$$\mathcal{A}(x, x^*) = \min\left\{1, \frac{p(x^*)q(x|x^*)}{p(x)q(x^*|x)}\right\}$$

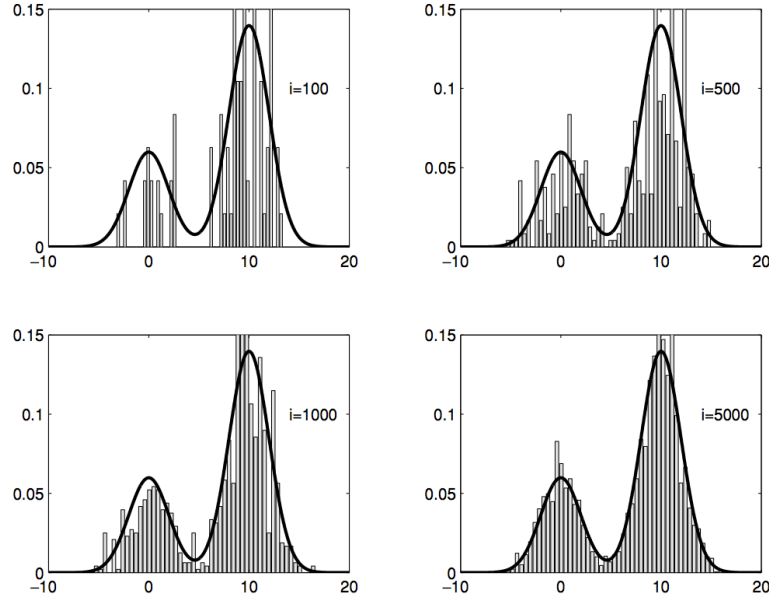


Figure 3.13 Target distribution and histogram of the MCMC samples at different iteration points.

otherwise it remains at x . The pseudo-code is shown in Figure 3.12, while Figure 3.13 shows the results of running the MH algorithm with a Gaussian proposal distribution

$$q(x^*|x(i)) = \text{Normal}(x^{(i)}, 100)$$

and a bimodal target distribution

$$p(x) \propto 0.3 \times \exp(0.2x^2) + 0.7\exp(0.2(x - 10)^2)$$

for 5000 iterations. As expected, the histogram of the samples approximates the target distribution.

The MH algorithm is very simple, but it requires careful design of the proposal distribution $q(x^*|x)$. In general, it is possible to use suboptimal inference and learning algorithms to generate data-driven proposal distributions.

The transition kernel for the MH algorithm is

$$K_{MH}(x^{(i+1)}|x^{(i)}) = q(x^{(i+1)}|x^{(i)})\mathcal{A}(x^{(i)}, x^{(i+1)}) + \sigma_{x^{(i)}}(x^{(i+1)})r(x^{(i)}),$$

where $r(x^{(i)})$ is the term associated with rejection

$$r(x^{(i)}) = \int_{\mathcal{X}} q(x^*|x^{(i)})(1 - \mathcal{A}(x^{(i)}, x^*))dx^*$$

It is fairly easy to prove that the samples generated by MH algorithm will mimic samples drawn from the target distribution asymptotically. By construction, K_{MH} satisfies the detailed balance condition

$$p(x^{(i)})K_{MH}(x^{(i+1)}|x^{(i)}) = p(x^{(i+1)})K_{MH}(x^{(i)}|x^{(i+1)})$$

and, consequently, the MH algorithm admits $p(x)$ as invariant distribution. To show that the MH algorithm converges, we need to ensure that there are no cycles (aperiodicity) and that every state that has positive probability can be reached in a finite number of steps (irreducibility). Since the algorithm always allows for rejection, it follows that it is aperiodic. To ensure irreducibility, we simply need to make sure that the support of $q(\cdot)$ includes the support of $p(\cdot)$.

The independent sampler and the Metropolis algorithm are two simple instances of the MH algorithm. In the independent sampler the proposal is independent of the current state, $q(x^*|x^{(i)}) = q(x^*)$. Hence, the acceptance probability is

$$\mathcal{A}(x^{(i)}|x^*) = \min\{1, \frac{p(x^*)q(x^{(i)})}{p(x^{(i)})q(x^*)}\}$$

The Metropolis algorithm assumes a symmetric random walk proposal $q(x^*|x^{(i)}) = q(x^{(i)}|x^*)$ and, hence, the acceptance ratio simplifies to

$$\mathcal{A}(x^{(i)}, x^*) = \min\{1, \frac{p(x^*)}{p(x^{(i)})}\}$$

3.4.3 From Samples to Probabilities

We can get traces of samples after the sampling algorithm. Probabilities can be estimated from a set of examples using the sample average. The sample average of a

proposition α is the number of samples where α is true divided by the total number of samples. The sample average approaches the true probability as the number of samples approaches infinity by the law of large numbers.

We treat the condition and the desired posterior as booleans. We will take a very simple query as example to illustrate how to calculate probability after sampling. Given a query

$$P(X = 1 | Y < 2)$$

we will get traces of samples where the samples all meet the condition $Y < 2$. The number of such samples generated is n . There are some other samples generated in the sampling process that can not meet the boolean requirement, which we will not count them in n . Then from these left n samples, we may find t samples that also meet the boolean requirement of $X = 1$. Thus the desired conditional probability would be t/n .

3.5 APIs for Other Languages

We leveraged the development tool SWIG (Simplified Wrapper and Interface Generator) Beazley *et al.* (1996) to make other languages be able to call for C functions, as we implemented the framework in C. SWIG is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby, and Tcl. It works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that scripting languages need to access the underlying C/C++ code. Once the user has the APIs, users are able to call the load probabilistic model function and inference function in each common language. That's how the portable is implemented.

The supported languages is listed as below:

Tcl	Python	Perl
Guile	Java	Ruby
Mzscheme	PHP	OCaml
C#	Pike	Chincken Scheme
Allegro CL	Modula-3	Lua
Lisp	R	Octave
Go	D	Javascript

No matter what domain languages we choose to use, the language to describe the probabilistic graphical model is the same, whose concrete syntax is showed in programming language syntax section, in Section 3.2. Below is an example to declare the flip model in our declaration language:

```

1  model flip_example() {
2      public flip
3      public x
4      private x1
5      private x2
6
7      flip ~ dbern(0.5)
8      x1 ~ dnorm(0, 1)
9      x2 ~ dgamma(1, 1)
10     x = if flip then x1 else x2 + 2
11 }

```

The challenge in this part is the C/C++ pointer doesn't exist in other languages like Java and Python. We cope with this problem by using wrapper with a specific form to hint the difference of the pointers and the normal variables.

We will give some examples of how our framework can be used in other common language like **C, Python, Java**.

3.5.1 C API

To encode the previous flip example in C, the APIs is showed as following:

```
1 #include <stdio.h>
2 #include "ppp.h"
3 #include <stdlib.h>
4 #include <time.h>
5
6 int main()
7 {
8     /* use pointers to structs because the client doesn't need to
9     know the struct sizes */
10    struct pp_state_t* state;
11    struct pp_instance_t* instance;
12    struct pp_query_t* query;
13    struct pp_trace_store_t* traces;
14    float result;
15
16    srand(time(NULL));
17    state = pp_new_state();
18    printf("> state created\n");
19
20    pp_load_file(state, "parse/models/flip.model");
21    printf("> file loaded\n");
22
23    instance = pp_new_instance(state, "flip_example", 0);
24    printf("> instance created\n");
25
26    query = pp_compile_query(instance, "x>2, x<3");
27    printf("> condition compiled\n");
28
29    traces = pp_sample(instance, query);
30    printf("> traces sampled\n");
31
32    query = pp_compile_query(instance, "flip==1");
33    printf("> query compiled\n");
```

```
33
34     pp_get_result(traces, query, &result);  /* "get_result" may not
35     be a good name */
36     printf("%f\n", result);
37
38     pp_infer(instance, "flip == 1", "x > 2, x < 3", &result);  /*
39     alternative way to get a result */
40     printf("%f\n", result);
41
42     pp_free(state);  /* free memory, associated models, instances,
43     queries, and trace stores are deallocated */
44
45     return 0;
46 }
```

3.5.2 Python API

If we want to encode the flip example in the same way in Python, we need to add an interface file which is the input to SWIG. The *swig* command produces a file “example_wrap.c” that should be compiled and linked with the rest of the program. In this case, we have built a dynamically loadable extension that can be loaded into the Python interpreter using the ‘load’ command. The interface file is showed in Figure 3.14.

```
1  /* libppp.i */
2  %module libppp
3  %include "typemaps.i"
4  %apply float *OUTPUT { float * result };
5  %{
6  /* Put header files here or function declarations like below */
7  #include "libppp.h"
8  %}
9
10 %include "libppp.h"
```

Figure 3.14 Interface file

where “libppp.h” is the original c header file containing the public APIs for user to load the model and make queries, which is showed in Figure 3.15.

```
1  struct pp_state_t;
2  struct pp_instance_t;
3  struct pp_query_t;
4  struct pp_trace_store_t;
5
6  struct pp_state_t* pp_new_state();
7  int pp_free(struct pp_state_t* state);
8
9  int pp_load_file(struct pp_state_t* state, const char* filename);
10 struct pp_instance_t* pp_new_instance(struct pp_state_t* state,
    const char* model_name, int* model_params);
11
12 struct pp_query_t* pp_compile_query(struct pp_instance_t* instance,
    const char* query_string);
13 struct pp_trace_store_t* pp_sample(struct pp_instance_t* instance,
    struct pp_query_t* query);
14 int pp_get_result(struct pp_trace_store_t* traces, struct
    pp_query_t* query, float* result);
15
16 // P(X|Y)
17 int pp_infer(struct pp_instance_t* instance, const char* X, const
    char* Y, float* result);
```

Figure 3.15 Header file of public APIs

And the API usage in Python in showed in Figure 3.16.

3.5.3 Java API

Similarly, a Java version for the flip example can be seen in Figure 3.17.

3.5.4 Ocaml API

If we want to use the model in Ocaml, we can use the API as showed in Figure 3.18.

For other languages, the APIs are similar. The interface file is just used to encode the public APIs of the framework and it is only used once to generate the wrapper of APIs

```
1 import libppp
2
3 state = libppp.pp_new_state()
4
5 libppp.pp_load_file(state, "../parse/models/flip.model")
6
7 instance = libppp.pp_new_instance(state, "flip_example", None)
8
9 query = libppp.pp_compile_query(instance, "x>2, x<3")
10
11 traces = libppp.pp_sample(instance, query)
12
13 query = libppp.pp_compile_query(instance, "flip==1")
14
15 success, result = libppp.pp_get_result(traces, query)
16 print("success:", success, "result:", result)
17
18 success, result = libppp.pp_infer(instance, "flip == 1", "x > 2, x <
19     3")
20 print("success:", success, "result:", result)
21 libppp.pp_free(state);
```

Figure 3.16 Python API usage example

in other languages. So users do not need to write the interface file on their own. They can directly write the queries in domain languages and load models in model files.

```
1 public class Flip {
2     public static void main(String[] args) {
3         System.setProperty("java.library.path", ".");
4         System.loadLibrary("libppp");
5         SWIGTYPE_p_pp_state_t state = libppp.pp_new_state();
6         SWIGTYPE_p_pp_instance_t instance;
7         SWIGTYPE_p_pp_query_t query;
8         SWIGTYPE_p_pp_trace_store_t traces;
9         int success;
10        float[] result = new float[2];
11
12        libppp.pp_load_file(state, "../parse/models/flip.model");
13
14        instance = libppp.pp_new_instance(state, "flip_example", null
15    );
16
17        query = libppp.pp_compile_query(instance, "x>2, x<3");
18
19        traces = libppp.pp_sample(instance, query);
20
21        query = libppp.pp_compile_query(instance, "flip==1");
22
23        success = libppp.pp_get_result(traces, query, result);
24        System.out.printf("result: d%", result);
25
26        success = libppp.pp_infer(instance, "flip == 1", "x > 2, x <
27    3", result);
28        System.out.printf("result: d%", result);
29
30        libppp.pp_free(state);
31    }
32 }
```

Figure 3.17 Java API usage example

```
1  open libppp
2
3  let flip_eg =
4      let state = Libppp.pp_new_state() in
5
6      Libppp.pp_load_file(state, "../parse/models/flip.model")
7
8      let instance = Libppp.pp_new_instance(state, "flip_example",
9      None) in
10
11      let query = Libppp.pp_compile_query(instance, "x>2, x<3") in
12
13      let traces = Libppp.pp_sample(instance, query) in
14
15      let query = Libppp.pp_compile_query(instance, "flip==1") in
16
17      let (success, result) = Libppp.pp_get_result(traces, query) in
18
19      let (success, result) = Libppp.pp_infer(instance, "flip == 1",
20      "x > 2, x < 3") in
21
22      Libppp.pp_free(state);
```

Figure 3.18 Ocaml API usage example

Chapter 4 Implementation

4.1 Implementation

The portable probabilistic programming framework is implemented in C and is consisted of four parts: parser, inference engine, query analyzer, and API implementations.

4.1.1 Parser

We implemented the parser based on the grammar in Chapter 3, Section 3.2. Since the language is used to describe Bayesian networks, we didn't generate abstract syntax tree. Instead, we generate the graph to describe the probabilistic graphical model with the following data structure:

```
1  struct BNVertex {
2      int type;          // draw or compute
3      float sample;     // last sampled value
4  };
5
6  struct BNVertexDraw {
7      struct BNVertex super; // extends BNVertex
8      int type;              // dbern, dnorm, dgamma, or constant
9  };
10
11 struct BNVertexDrawBern {
12     struct BNVertexDraw super;
13     float p;
14 };
15
16 struct BNVertexDrawNorm {
17     struct BNVertexDraw super;
18     float mean;
19     float variance;
20 };
```

```
21
22 struct BNVertexDrawGamma {
23     struct BNVertexDraw super;
24     float a;
25     float b;
26 };
27
28 struct BNVertexDrawConst {
29     struct BNVertexDraw super;
30     float c;
31 };
32
33 struct BNVertexCompute {
34     struct BNVertex super; // extends BNVertex
35     int type; // + or if
36 };
37
38 struct BNVertexComputePlus {
39     struct BNVertexCompute super;
40     struct BNVertex* left;
41     struct BNVertex* right;
42 };
43
44 struct BNVertexComputeIf {
45     struct BNVertexCompute super;
46     struct BNVertex* condition;
47     struct BNVertex* consequent;
48     struct BNVertex* alternative;
49 };
```

The generated network is (1) constructed, (2) topological sorted, and (3) stored in an array `struct BNVertex* vertices[]`. The inference engine can traverse this array from the start to the end, and access sample values stored in the vertices. Because the network is already topological sorted, it is guaranteed that inference engine always


```

1  float gaussian(float mu, float sigma)
2  {
3      float u, v, x, y, q;
4      do {
5          u = 1 - randomC();
6          v = 1.7156 * (randomC() - 0.5);
7          x = u - 0.449871;
8          y = fabs(v) + 0.386595;
9          q = x * x + y * (0.196 * y - 0.25472 * x);
10     } while (q >= 0.27597 && (q > 0.27846 || v * v > -4 * u * u *
11     log(u)));
12     return mu + sigma * v / u;
13 }

```

Figure 4.1 Sampling Gaussian distribution

reads the correct sample values.

4.1.2 Probabilistic Distributions

The implementations of the probabilistic distributions are generally based on rejection sampling. We leveraged the algorithms in probability theory to generate variables from a specified distribution.

Figure 4.1 is an example to show how we derive the samples of Gaussian distribution from `randomC` distribution which generate random values in $[0, 1]$:

4.1.3 Inference Engine

In the implementation of the MH algorithm in our inference engine, we leverage the approach proposed by Wingate et al. (2011), which memorized trace information and update trace iteratively. More specifically, Let a database \mathbb{D} be defined as a mapping $\mathbb{N} \rightarrow \mathcal{T} \times X \times \mathbb{L} \times \theta$, where \mathbb{N} is the name of a random choice, \mathcal{T} is its elementary random primitives(ERP) type, which represents a sample function of distributions like `rand()`, `gaussian()`, X is its value, θ are ERP parameters, and \mathbb{L} is this random value's likelihood. Missing entries are allowed here.

We can control the execution of a function f' by controlling the values in \mathbb{D} . When

Algorithm 2 An MCMC trace sampler.

```

1: Initialize:  $[ll, \mathbb{D}] = \text{trace\_update}(\emptyset)$ 
2: Repeat forever:
3:   Select a random  $f_k$  via its name  $n$ 
4:   Look up its current value  $(t, x, l, \theta_{db}) = \mathbb{D}(n)$ .
5:   Propose a new value  $x' \sim \mathcal{K}_t(\cdot | x, \theta_{db})$ 
6:   Compute  $F = \log \mathcal{K}_t(x' | x, \theta_{db})$ 
7:   Compute  $R = \log \mathcal{K}_t(x | x', \theta_{db})$ 
8:   Compute  $l' = \log p_t(x' | \theta_{db})$ 
9:   Let  $\mathbb{D}' = \mathbb{D}$ 
10:  Set  $\mathbb{D}'(n) = (t, x', l', \theta_{db})$ 
11:   $[ll', \mathbb{D}'] = \text{trace\_update}(\mathbb{D}')$ ;
12:  if (  $\log(\text{rand}) < ll' - ll + R - F$  )
13:    // accept
14:     $\mathbb{D} = \mathbb{D}'$ 
15:     $ll = ll'$ 
16:    // clean out unused values from  $\mathbb{D}$ 
17:  else
18:    // reject; discard  $\mathbb{D}'$ 
19:  endif;
20: end repeat;
```

Figure 4.2 An MCMC trace sampler.

Algorithm 3 Function trace_update(\mathbb{D})

```

1: Set  $ll = 0$ 
2: Execute  $f$ :
3: For all random choices  $k$ :
4:   Run computation until choice  $k$ ,
   determining  $n, t_c, \theta_c$  for  $k$ .
5:   Look up  $(t, x, l, \theta_{db}) = \mathbb{D}(n)$ 
6:   if a value is found in the database and  $t = t_c$ 
7:     if parameters match (ie,  $\theta_c == \theta_{db}$ )
8:        $ll = ll + l$ 
9:     else
10:      // rescore ERP with new parameters
11:      compute  $l = \log p_{t_c}(x|\theta_c)$ 
12:      store  $\mathbb{D}(n) = (t_c, x, l, \theta_c)$ 
13:       $ll = ll + l$ 
14:    endif
15:  else
16:    // sample new randomness
17:    sample  $x \sim p_t(\cdot|\theta_c)$ 
18:    compute  $l = \log p_t(x|\theta_c)$ 
19:    store  $\mathbb{D}(n) = (t_c, x, l, \theta_c)$ 
20:     $ll = ll + l$ 
21:  endif
22:  Set return value of  $f_k$  to  $x$ 
23: end for all
24: return [  $ll, \mathbb{D}$  ]

```

Figure 4.3 Algorithm for trace update

f' encounters an f'_k , it computes its name $n \in \mathbb{N}$, its parameters θ_c (where c is a mnemonic for “current”), and its current type $t_c \in \mathcal{T}$. It then looks up $(t, x, l, \theta_{db}) = \mathbb{D}(n)$. If a value is found and the types match, we set the return value of f_k to be x . Otherwise, the value is sampled from the appropriate ERP $x \sim p_{t_c}(\cdot | \theta_c)$, its likelihood is computed, and the corresponding entry in the database is updated. This is formalized in the `trace_update` procedure, defined in Figure 4.3.

We use `trace_update` to define our overall MCMC algorithm, shown in Figure 4.2. Given a current trace x and score $p(x)$, we proceed by reconsidering one random choice x_k . We equip each ERP type with a proposal kernel $\mathcal{K}_t(x'|x, \theta)$, which we use to generate proposals to x_k . After a proposal, we call `trace_update` to generate a new trace x' , and compute its likelihood $p(x')$, which is the product of any reused random choices, and any new randomness that was sampled. This gives us an overall score, which is used as an MH accept ratio

$$\alpha = \min\left\{1, \frac{p(x')\mathcal{K}_t(x|x, \theta)}{p(x)\mathcal{K}_t(x'|x, \theta)}\right\}$$

There are some other methods to implement an inference engine such as the *non-standart interpretations* proposed by Wingate et al. (2011), which showed how *nonstandard interpretations* of probabilistic programs can be used to perform efficient inference algorithms. In their method, information about the structure of the distributions (which is the dependencies or gradients in the probabilistic graphical models) is derived as monad-like side computation as the same time of executing the program. Meanwhile, the interpretations can be coded easily with some special-purpose objects and operator overloading. They promoted the inference efficiency performance by using the structure information of distribution as part of the variety of inference algorithms.

Additionally, because of the program is in a machine-readable form, various of techniques from compiler design and program analysis can be used in the inference engine. Gordon et al. (2013) designed a new model-learner pattern for Bayesian reasoning. In their work, a new probabilistic programming abstraction was proposed, a typed Bayesian model, based on a pair of probabilistic expressions for the prior and sampling

samples(times)	accuracy	time(s)
100	0.7843125	0.07
500	0.794375	0.059
1000	0.986325	0.055
2000	0.98115	0.057
5000	0.9480125	0.079
8000	0.9818375	0.101
10000	0.9710375	0.113
20000	0.9972125	0.169
50000	0.9870875	0.363
80000	0.9563875	0.562
100000	0.992675	0.691

Figure 4.4 Table of flip example

distributions. Also, Claret et al. (2013) presented a new algorithm for Bayesian inference over probabilistic programs, which is based on data flow analysis techniques from the program analysis community. These can be improvements to our framework in the future development.

4.1.4 API

To enable the APIs for other programming languages, we first compiled the interface file as showed in Section 3.5 and generated the wrapper code that other languages need to access the underlying C/C++ code. Then the users will have access to the APIs of the portable probabilistic programming language framework in their domain language.

4.2 Case Study

Following the example we have illustrated above: flip example, which is showed in Figure 3.2, we studied the accuracy and time cost according to the different assignments of number of samples. The result is showed in Figure 4.4 and Figure 4.5.

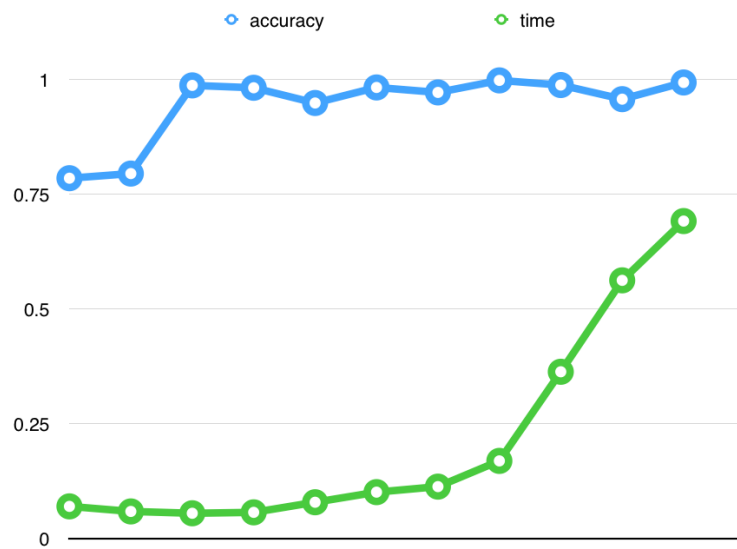


Figure 4.5 Flip example: accuracy and time according to the number of samples.

Chapter 5 Conclusion

We designed and built a portable probabilistic programming framework which can:

1. Describe Bayesian networks.
2. Automatically infer conditional probability.
3. Embedded into every common programming language to address the cross-platform problem

Different from all the other probabilistic programming languages or systems as discussed in Chapter 2, the design of our framework didn't extend any domain programming language, but allow users to use in many programming languages, which is portable. We have designed the syntax of the portable probabilistic programming language which can declare Bayesian networks and allow users to query condition probability in domain language. Our framework can benefit users to develop in cross-platform as we offered APIs for each common used programming language. In this way, developers don't need to get the hang of several different probabilistic programming languages which are extended from different domain languages. They can use one language to declare their Bayesian networks while query the models in desired domain developing languages.

The goal of probabilistic programming is to help the large number of programmers who have domain expertise, but lack of expertise in machine learning. It allows programmers to put more emphasis on the design of a model rather than spending lots of time implementing the graphical models and inference tasks. In Probabilistic programming, modeling and inference have been disentangled. The research in probabilistic programming lies in the intersection of artificial intelligence, stochastics and programming language. The challenges of this framework lies in the design of the portable probabilistic programming language and the implementation of the inference engine, which can automatically do inference.

The main directions for improvement are better mixing and faster inference. Goodman (2013). The options of inference algorithms can be enlarged to target more generative models. What's more, for difference models, ideally the inference engines can choose the most suitable and efficient inference method thus to enhance the performance of inference engine. Gershman and Goodman argued that the brain operates in the setting of amortized inference, where numerous related queries must be answered. Thus they proposed a form of exible reuse, according to which shared inferences are cached and composed together to answer new queries. There is also a way to work on parallel processing of multiple chains and try to devise thus to implement more efficient and robust inference engine.

REFERENCE

- [1] LUNN D J, THOMAS A, BEST N, et al. WinBUGS-a Bayesian modelling framework: concepts, structure, and extensibility[J]. *Statistics and computing*, 2000, 10(4):325–337.
- [2] GOODMAN N, MANSINGHKA V, ROY D, et al. Church: a language for generative models[J]. *arXiv preprint arXiv:1206.3255*, 2012.
- [3] MCCALLUM A, SCHULTZ K, SINGH S. FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs.[C]//NIPS. .[S.l.]: [s.n.] , 2009:1249–1257.
- [4] WANG S S, WAND M P. Using Infer. NET for statistical analyses[J]. 2011.
- [5] HERSHEY S, BERNSTEIN J, BRADLEY B, et al. Accelerating Inference: towards a full Language, Compiler and Hardware stack[J]. *arXiv preprint arXiv:1212.2991*, 2012.
- [6] BEAZLEY D M, et al. SWIG: An easy to use tool for integrating scripting languages with C and C++[C]//Proceedings of the 4th USENIX Tcl/Tk workshop. .[S.l.]: [s.n.] , 1996:129–139.
- [7] GOODMAN N D. The principles and practice of probabilistic programming[C]//ACM SIGPLAN Notices. .[S.l.]: [s.n.] , 201348:399–402.
- [8] GORDON A D, HENZINGER T A, NORI A V, et al. Probabilistic programming[C]//International Conference on Software Engineering (ICSE, FOSE track). .[S.l.]: [s.n.] , 2014.
- [9] KOLLER D, FRIEDMAN N. Probabilistic graphical models: principles and techniques[M].[S.l.]: MIT press, 2009.
- [10] HECKERMAN D, GEIGER D, CHICKERING D M. Learning Bayesian networks: The combination of knowledge and statistical data[J]. *Machine learning*, 1995, 20(3):197–243.
- [11] HECKERMAN D. A tutorial on learning with Bayesian networks[M].[S.l.]: Springer, 1998.

- [12] KINDERMANN R, SNELL J L, et al. Markov random fields and their applications[M], Vol. 1.[S.l.]: American Mathematical Society Providence, RI, 1980.
- [13] ANDRIEU C, DE FREITAS N, DOUCET A, et al. An introduction to MCMC for machine learning[J]. Machine learning, 2003, 50(1-2):5–43.
- [14] NEAL R M. Slice sampling[J]. Annals of statistics, 2003:705–741.
- [15] CHIB S, GREENBERG E. Understanding the metropolis-hastings algorithm[J]. The American Statistician, 1995, 49(4):327–335.
- [16] PAIGE B, WOOD F. A Compilation Target for Probabilistic Programming Languages[J]. arXiv preprint arXiv:1403.0504, 2014.
- [17] PFEFFER A. IBAL: A Probabilistic Rational Programming Language[C]//In Proc. 17th IJCAI.[S.l.]: Morgan Kaufmann Publishers, 2001:733–740.
- [18] POON H, DOMINGOS P. Joint unsupervised coreference resolution with Markov logic[C]//Proceedings of the conference on empirical methods in natural language processing. .[S.l.]: [s.n.] , 2008:650–659.
- [19] BISHOP C M, et al. Pattern recognition and machine learning[M], Vol. 1.[S.l.]: springer New York, 2006.
- [20] WINGATE D, STUHLMÜLLER A, GOODMAN N D. Lightweight implementations of probabilistic programming languages via transformational compilation[C]//International Conference on Artificial Intelligence and Statistics. .[S.l.]: [s.n.] , 2011:770–778.
- [21] WINGATE D, GOODMAN N D, STUHLMUELLER A, et al. Nonstandard Interpretations of Probabilistic Programs for Efficient Inference.[C]//NIPS. .[S.l.]: [s.n.] , 2011:1152–1160.
- [22] GORDON A D, AIZATULIN M, BORGSTROM J, et al. A model-learner pattern for Bayesian reasoning[C]//ACM SIGPLAN Notices. .[S.l.]: [s.n.] , 201348:403–416.
- [23] CLARET G, RAJAMANI S K, NORI A V, et al. Bayesian inference using data flow analysis[C]//ESEC/SIGSOFT FSE. .[S.l.]: [s.n.] , 2013:92–102.
- [24] BOX G E, MULLER M E, et al. A note on the generation of random normal deviates[J]. The annals of mathematical statistics, 1958, 29(2):610–611.

- [25] MARSAGLIA G, TSANG W W. The ziggurat method for generating random variables[J]. Journal of Statistical Software, 2000, 5(8):1–7.
- [26] CASELLA G, ROBERT C P. Monte Carlo statistical methods[].[S.l.]: Springer-Verlag, New York.
- [27] GERSHMAN S J, GOODMAN N D. Amortized Inference in Probabilistic Reasoning[J].

Acknowledgements

I take this opportunity to express my profound gratitude and deep regards to my supervisor Prof. Kenny Q. Zhu for his exemplary guidance, monitoring and constant encouragement throughout the course of this thesis. The blessing, help and guidance given by him time to time shall carry me a long way in the journey of life on which I am about to embark.

I also feel grateful for the constructive suggestions and comments from the students in our lab (ADAPT Lab), including Xiao Jia, Qianjing Song, Jessie Luo, Jacky Jiang, Kangqi Luo and Edward He, which helped me in completing this task through various stages.

I offer my sincere appreciation for the learning opportunities provided by Computer Science Department, where the research was undertaken. And great thanks to my teachers in Shanghai Jiao Tong University, who not only delivered knowledge, but also inspired the students' talent and set role models on characteristics.

I would like to thank Shanghai Jiao Tong University Chun-Tsung Program committees for supporting and granting this project.

I also want to take this opportunity to offer my special thanks to Prof. Liqing Zhang for inspiring my passion in computer science and research interest.

Lastly, I thank almighty, my parents, and friends for their constant encouragement without which the thesis would not be possible.