

Composable Probabilistic Inference with BLAISE

by

Keith Allen Bonawitz



Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

© Massachusetts Institute of Technology 2008. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
April 30, 2008

Certified by
Patrick H. Winston
Ford Professor of Artificial Intelligence and Computer Science
Thesis Supervisor

Certified by
Joshua B. Tenenbaum
Paul E. Newton Career Development Professor
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Students



Composable Probabilistic Inference with BLAISE

by

Keith Allen Bonawitz



Submitted to the Department of Electrical Engineering and Computer Science
on April 30, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

If we are to understand human-level cognition, we must understand how the mind finds the patterns that underlie the incomplete, noisy, and ambiguous data from our senses and that allow us to generalize our experiences to new situations. A wide variety of commercial applications face similar issues: industries from health services to business intelligence to oil field exploration critically depend on their ability to find patterns in vast amounts of data and use those patterns to make accurate predictions.

Probabilistic inference provides a unified, systematic framework for specifying and solving these problems. Recent work has demonstrated the great value of probabilistic models defined over complex, structured domains. However, our ability to imagine probabilistic models has far outstripped our ability to programmatically manipulate them and to effectively implement inference, limiting the complexity of the problems that we can solve in practice.

This thesis presents BLAISE, a novel framework for composable probabilistic modeling and inference, designed to address these limitations. BLAISE has three components:

- The **BLAISE State-Density-Kernel (SDK)** graphical modeling language that generalizes factor graphs by: (1) explicitly representing inference algorithms (and their locality) using a new type of graph node, (2) representing hierarchical composition and repeated substructures in the state space, the interest distribution, and the inference procedure, and (3) permitting the structure of the model to change during algorithm execution.
- A suite of **SDK graph transformations** that may be used to extend a model (e.g. to construct a mixture model from a model of a mixture component), or to make inference more effective (e.g. by automatically constructing a parallel tempered version of an algorithm or by exploiting conjugacy in a model).
- The **BLAISE Virtual Machine**, a runtime environment that can efficiently execute the stochastic automata represented by BLAISE SDK graphs.



BLAISE encourages the construction of sophisticated models by composing simpler models, allowing the designer to implement and verify small portions of the model and inference method, and to reuse model components from one task to another. BLAISE decouples the implementation of the inference algorithm from the specification of the interest distribution, even in cases (such as Gibbs sampling) where the shape of the interest distribution guides the inference. This gives modelers the freedom to explore alternate models without slow, error-prone reimplementations. The compositional nature of BLAISE enables novel reinterpretations of advanced Monte Carlo inference techniques (such as parallel tempering) as simple transformations of BLAISE SDK graphs.

In this thesis, I describe each of the components of the BLAISE modeling framework, as well as validating the BLAISE framework by highlighting a variety of contemporary sophisticated models that have been developed by the BLAISE user community. I also present several surprising findings stemming from the BLAISE modeling framework, including that an Infinite Relational Model can be built using exactly the same inference methods as a simple mixture model, that constructing a parallel tempered inference algorithm should be a point-and-click/one-line-of-code operation, and that Markov chain Monte Carlo for probabilistic models with complicated long-distance dependencies, such as a stochastic version of Scheme, can be managed using standard BLAISE mechanisms.

Thesis Supervisor: Patrick H. Winston

Title: Ford Professor of Artificial Intelligence and Computer Science

Thesis Supervisor: Joshua B. Tenenbaum

Title: Paul E. Newton Career Development Professor

Acknowledgments

This thesis would not have been possible without the help and support of innumerable friends, family and colleagues. Let me start by thanking my thesis committee: Patrick Winston, Josh Tenenbaum, and Antonio Torralba. Patrick has advised me since I was an undergraduate, guiding me through the deep questions of artificial intelligence from the perspective of understanding human-level cognition. I am also deeply indebted to Patrick for teaching me both how to frame my ideas and how to present them to others – without a doubt, Patrick is a master of this art, and I can only hope I have absorbed some of his skills. I thank Josh for teaching me the power of combining probabilistic modeling and symbolic reasoning. Moreover, I thank Josh for fostering a vibrant and social research environment, both inside the lab and out – there is definitely something wonderful to be said for insightful conversation while hiking in New Hampshire.

I cannot praise my community at MIT highly enough. My friends and colleagues in Patrick’s lab, in Josh’s lab, and throughout the Institute have been the most wonderful support system I could ask for, not to mention their influence on my thoughts about this work and science in general. Special thanks to Dan Roy for bravely using a very early version of this work for his own research, despite its rough edges and lack of documentation, and to Vikash Mansinghka, Beau Cronin, and Eric Jonas for embracing the ideas in this thesis and providing crucial input and endless encouragement.

I thank my parents, Lynn and John Bonawitz, for teaching me to follow my dreams with passion and integrity, and to build the life that works for me. They and my brother, Michael Bonawitz, are remarkable in their ability to always let me know how much I’m loved and missed, while still encouraging my pursuits – even when those pursuits take me far away from them.

But most of all, this thesis would never have happened if not for my wife Liz Bonawitz. Liz’s boundless love and support have been a constant source of strength, encouraging me to be the best I can be, both personally and professionally, and

challenging me to forge my own path and chase both my ideas and my ideals. From graduate school applications through my defense, Liz has been by my side every step of the way – introducing me to many of the people and ideas that have become the core of my work, but also making sure I didn't miss out on the fun and laughter life has to offer.

This material is based upon work supported by the National Science Foundation under Grant No. 0534978. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.



Contents

1	Introduction	15
1.1	Thesis statement and organization	18
2	Background: Monte Carlo Methods for Probabilistic Inference	21
2.1	Probabilistic Models and Inference	22
2.2	Approximate Inference	24
2.3	Monte Carlo Methods	26
2.4	Markov Chain Monte Carlo	27
2.5	Transdimensional MCMC	29
2.6	Tempered Inference	30
2.7	Particle Filtering	31
3	The BLAISE State–Density–Kernel Graphical Modeling Language	35
3.1	An overview of the BLAISE modeling language	36
3.2	BLAISE States	39
3.3	BLAISE Densities	45
3.4	BLAISE Kernels	49
3.4.1	Hybrid Kernels	55
3.4.2	Metropolis-Hastings and Gibbs Sampling	58
3.5	Densities for state spaces of unknown dimensionality or variable structure	64
3.6	Kernels for state spaces of unknown dimensionality or variable structure	66
3.7	Compositionality and state space constraints	69
3.8	Initialization Kernels	73

3.9	Comparison to the Model-View-Controller architecture	79
3.10	Comparison to reactive programming languages	79
3.11	Comparison to Infinite Contingent Bayes Nets	81
3.12	Discussion	82
4	BLAISE Transformations	85
4.1	An introduction to BLAISE transformations	85
4.2	Tempered models	87
4.2.1	The temper transform	87
4.2.2	Simulated Annealing	88
4.2.3	Parallel Tempering	89
4.3	Particle Filtering	92
4.4	Hybrid Algorithms	93
4.5	Mixture Models	94
4.5.1	Parametric mixture models with fixed size and fixed weights .	96
4.5.2	Parametric mixture models with fixed size and unknown weights	99
4.5.3	Parametric mixture models of unknown size	100
4.5.4	Non-parametric mixture models	101
4.5.5	Bridged-form mixture models	102
4.6	Conjugate models	103
4.7	Discussion	105
5	The BLAISE Virtual Machine	107
5.1	An introduction to the BLAISE Virtual Machine	107
5.2	States are mutated in-place	109
5.3	Density evaluations are memoized	109
5.4	Tree-structured Transactional Caching for States and Densities	111
5.5	Incremental Updates for Efficient Collection Densities	115
5.6	State, Density, and Kernel responses to State modifications	117
5.7	Discussion	119

6	Applications in BLAISE	121
6.1	Generative Vision	122
6.2	Analysis of Neurophysiological Data	127
6.3	Relational Models	131
6.3.1	Infinite Relational Model	134
6.3.2	Annotated Hierarchies	141
6.4	Latent Dirichlet Allocation	143
6.5	Standard Graphical Models in BLAISE	148
6.6	Systematic Stochastic Search	151
6.7	Higher-level probabilistic programming languages	156
6.7.1	BUGS on BLAISE	157
6.7.2	Church: A stochastic lambda calculus	159
6.8	Discussion	162
7	Related Work	165
7.1	Bayes Net Toolbox	165
7.2	BUGS	167
7.3	VIBES	167
7.4	BLOG: Bayesian Logic	168
7.5	IBAL	168
8	Conclusion	169
8.1	New perspectives	169
8.2	Enabled research	170
8.3	Contributions	173
A	Conditional Hybrid Kernels	177
A.1	An Introduction to Conditional Hybrid Kernels	177
A.2	Conditional Hybrid Kernel Stationary Distribution Proof	179
A.3	Truncated Stationary Distribution Lemma	181
B	BLAISE SDK Legend	183



List of Figures

3-1	Preview of the BLAISE modeling language compared to other standard probabilistic modeling languages	37
3-2	A beta-binomial model in factor graph notation	40
3-3	The state space for a beta-binomial model in factor graph notation and BLAISE SDK notation	40
3-4	Simple plated factor graphs and the equivalent BLAISE State structures	41
3-5	Beta-binomial model for multiple data points, in plated factor graph notation and BLAISE SDK notation	42
3-6	Mixture models in Bayes net notation and BLAISE notation	44
3-7	The state space and interest distribution for a single-datapoint beta-binomial model in factor graph notation and BLAISE SDK notation	45
3-8	The state space and interest distribution for a multiple-datapoint beta-binomial model in factor graph notation and BLAISE SDK notation	46
3-9	A beta-binomial BLAISE model with Kernels	51
3-10	Kernels for mixture models	52
3-11	Operations supported by Kernels	54
3-12	Operations supported by Moves	55
3-13	A Gaussian (normal) BLAISE model with Kernels	57
3-14	Metropolis-Hastings Kernels	59
3-15	Pseudocode for a Metropolis-Hastings Kernel's SAMPLE-NEXT-STATE.	60
3-16	Kernels for mixture models	61
3-17	Pseudocode for a Gibbs sampling Kernel's SAMPLE-NEXT-STATE.	61

3-18	Blocking in Metropolis-Hastings Kernels	62
3-19	Blocking in Gibbs Kernels	63
3-20	A mixture model, demonstrating Associated Collection Densities . . .	65
3-21	A mixture model, demonstrating Virtual Hybrid Kernels	67
3-22	Virtual Hybrid Kernels can be analyzed as a Conditional Hybrid of Concrete Cycles	68
3-23	Constraint States in “bridged form” mixture models	71
3-24	Bridged form mixture models with Kernels	72
3-25	Initialization Kernels	74
3-26	Initialization Kernels for Bridged Mixtures	75
3-27	Non-parametric multi-feature beta-binomial mixture model	78
3-28	Model-View-Controller design pattern and BLAISE SDK analogy . . .	80
4-1	The <i>temper</i> transform	87
4-2	The <i>simulated annealing</i> transform	88
4-3	The <i>parallel tempering</i> transform	89
4-4	The <i>particle filtering</i> transform	91
4-5	Parallel tempered reversible jump Markov chain Monte Carlo	94
4-6	Particle filtering mixed with MCMC	95
4-7	Particle filtering with simulated annealing	96
4-8	Parallel tempered particle filter	97
4-9	Parametric mixture model transform (fixed size, fixed weights)	98
4-10	Parametric mixture model transform (fixed size, unknown weights) .	99
4-11	Parametric mixture model transform (variable size)	100
4-12	Non-parametric mixture model transform	101
4-13	Beta-binomial conjugacy transform	103
5-1	Timing comparison for State mutation vs. Copy-On-Write	110
5-2	Timing comparison for transactions and memoization	112
5-3	Timing comparison for transactions and memoization, experiment 2 .	113

5-4	BLAISE uses a tree-structured transactional caching system to enhance performance.	115
5-5	“Let” Kernels in the Virtual Machine	118
6-1	Generative Vision input	124
6-2	Generative Vision SDK	125
6-3	Generative Vision requires parallel tempering	126
6-4	Functional model for a V1 neuron	127
6-5	LNP model of a V1 neuron, as a Bayes net	128
6-6	LNP model of a V1 neuron, as a BLAISE model	129
6-7	Generalized LNP model of a V1 neuron, as a BLAISE model	130
6-8	Neurophysiological experimental data and model predictions generated using BLAISE (from Schummers et al. [57])	131
6-9	Input and output for the Infinite Relational Model	135
6-10	Multi-feature Mixture Model SDK (reprise)	136
6-11	Infinite Relational Model SDK	137
6-12	Infinite Relational Model applied to a synthetic dataset	138
6-13	Infinite Relational Model applied to Movielens	139
6-14	BLAISE IRM scales near linearly in number of datapoints	140
6-15	Relational data modeled with Annotated Hierarchies (from Roy et al. [54])	142
6-16	Latent Dirichlet Allocation Bayes Net	143
6-17	Latent Dirichlet Allocation Model SDK	144
6-18	LDA Wikipedia input	146
6-19	LDA Wikipedia learned topics	147
6-20	The Ising model as a factor graph and as a BLAISE model	149
6-21	Kernels for factor graphs	150
6-22	Bayes nets can be reduced to factor graphs	151
6-23	Systematic Stochastic Search: Sequentialize transform	154
6-24	Systematic Stochastic Search for stereo vision	155

6-25	A standard example of a BUGS model	157
6-26	A non-parametric mixture model in a hypothetical BUGS extension .	158
6-27	The Infinite Relational Model as a Church program	160
6-28	The Infinite Hidden Markov Model as a Church program	161
7-1	Software package comparison matrix	166
8-1	BLAISE software and hardware abstraction stack	171

Chapter 1

Introduction

My thesis is that a framework for probabilistic inference can be designed that enables efficient composition of both models and inference procedures, that is suited to the representational needs of emerging classes of probabilistic models, and that supports recent advances in inference.

Probabilistic inference is emerging as a common language for computational studies of the mind. Cognitive scientists and artificial intelligence researchers are appealing more frequently to probabilistic inference in both normative and descriptive accounts of how humans can draw confident conclusions from incomplete or ambiguous evidence [9, 60]. Explorations of human category learning [61], property induction [29], and causal reasoning [23] have all found remarkable accord between human performance and the predictions of Bayesian probabilistic models. Probabilistic models of the human visual system help us understand how top-down and bottom-up reasoning can be integrated [67]. Computational neuroscientists are even finding evidence that probabilistic inference may help explain the behavior of individual neurons and neuronal networks [37, 52]. Beyond studying the mind, much practical use is found for probabilistic models in fields as diverse as business intelligence [48], bioinformatics, and medical informatics [27].

Probabilistic models are not the only way to approach problems of reasoning under uncertainty, but they have recently exploded in popularity for a number of reasons.

First, Bayesian probabilistic modeling encourages practitioners to be forthright with their assumptions. As phrased by MacKay [38] (p. 26), “you cannot do inference without making assumptions.” All inference techniques make assumptions about how unobserved (or future) values are related to observable (or past) values; however, non-Bayesian models typically leave these assumptions implicit. As a result, it is difficult both to evaluate the justification of the assumptions and to change those assumptions. In contrast, Bayesian models are explicit about their prior assumptions. Another virtue of probabilistic models is that, given a few intuitive desiderata for reasoning under uncertainty, probability theory is the *unique* calculus for such reasoning (this result is known as Cox’s Theorem [28]). This provides the modeler with assurance that the mathematical framework in which his models are embedded is capable of correctly handling future model extensions and provides a common language for the interchange of modeling results. The use of the term “language” here is non-accidental: probability theory provides all the elements of a powerful programming or engineering language: primitives (e.g., random variables and simple conditional distributions), means of combination (e.g., joint distributions composed from simple conditional distributions that may share variables), and means of abstraction (e.g., marginalizing out variables to produce new conditional distributions) [2].

In addition to these general features of probability theory, a confluence of factors are contributing to a renaissance of probabilistic modeling in cognitive science. One of these is the détente between practitioners of structured symbolic reasoning and statistical inference. This has resulted in the investigation of “sophisticated” (cf. [9]) probabilistic models, in which the random variables have structured representations for their domains, such as trees, graphs, grammars, or logics. For example, [23] treat learning a causal structure as a probabilistic inference problem including a random variable on the domain of causal networks. This variable is conditioned on a more abstract structure encoding a simple probabilistic grammar for possible causal networks. Probabilistic inference on this hierarchical layering of structured representations, applied to (for example) data about behaviors, diseases, and symptoms, allows one to learn not only a reasonable causal network for a particular set

of observations, but also more abstract knowledge such as “diseases cause symptoms, but symptoms never cause behaviors.” As another example of sophisticated models, [54] learn a classification hierarchy from a set of features and relations among objects while simultaneously learning what level of the hierarchy is most appropriate for predicting each feature and relation in the dataset; this work can be viewed as a probabilistic extension of hierarchical semantic networks [10]. These examples illustrate how incorporating the representational power of symbolic systems has enabled the inferential power of probability theory to be brought to bear on problems that not long ago were thought to be outside the realm of statistical models, resulting in robust symbolic inference with principled means for balancing new experience with prior beliefs.

Contemporary probabilistic inference problems are also sophisticated in their use of advanced mathematics throughout modeling and inference. Nonparametric models such as the Chinese Restaurant Process (or Dirichlet Process) [6, 30] and the Indian Buffet Process [24] allow the dimensionality of a probabilistic model to be determined by the data being explained; in effect, the model grows with the data. Nonparametrics are the basis of many recent cognitive models such as the Infinite Relational Model [30], in which relational data is explained by partitioning the objects in each domain into a set of classes, each of which behaves homogeneously under the relation. The number of classes in each domain is not known *a priori*, and instead a Chinese Restaurant Process is used to allow the Infinite Relational Model to use just as many classes as the data justifies. Along with advanced mathematics for modeling come advanced techniques for performing inference on these models. For example, when performing inference in a nonparametric model, special care must be taken to account for the variable dimensionality of the model. For approximate inference techniques based on Markov chain Monte Carlo (MCMC), this special care takes the form of Reversible Jump MCMC [21] and involves the computation of a Jacobian factor relating parameter spaces of different dimension. Reversible Jump MCMC ensures only the correctness of inference; other advanced techniques are focused on making inference tractable in models of increasing complexity, whether that complexity is due to

complexly structured random variables (e.g., with domains such as the space of all graphs), hierarchically layered models (e.g., causal structures and causal grammars as in [23]), or models with unknown dimensionality (e.g. resulting from the use of non-parametrics). Examples of advanced techniques for improving inference performance include parallel tempering [15], in which probabilistic inference on easier versions of the probabilistic model is used to guide inference in the desired full-difficulty model, and sequential methods such as particle filtering [14], an online, population-based Monte Carlo method that only uses each datapoint once, at the time when it arrives online.

Unfortunately, existing tools are not designed to handle the kind of sophisticated models and inference techniques that are required today. As a result, most modelers currently construct their own special purpose implementations of these algorithms for every model they create — an inefficient and error-prone process which frequently leads the practitioner to forgo many advanced techniques due to the difficulty of implementing them in a system that does not offer use of the proper abstractions.

1.1 Thesis statement and organization

My thesis is that a framework for probabilistic inference can be designed that enables efficient composition of both models and inference procedures, that is suited to the representational needs of emerging classes of probabilistic models, and that supports recent advances in inference.

Chapter 2 reviews the mathematical underpinnings of probabilistic inference.

Chapters 3–6 directly address the claims in my thesis statement. Chapter 3 introduces the BLAISE State–Density–Kernel (SDK) graphical modeling language and shows how this language supports composition of models and inference procedures. Chapter 4 highlights how several recent advances in inference are supported by interpreting them as graph transformations in the SDK language. Chapter 5 describes the BLAISE virtual machine, which can efficiently execute the stochastic automata described by BLAISE SDK graphs. Chapter 6 describes several applications involv-

ing emerging classes of probabilistic models, each of which has been built using the BLAISE framework.

With the thesis supported, chapter 7 compares BLAISE to existing probabilistic inference frameworks, and chapter 8 reviews the contributions this thesis makes to the field.



Chapter 2

Background: Monte Carlo

Methods for Probabilistic Inference

This thesis focuses on Monte Carlo methods for probabilistic inference, a class of algorithms for drawing conclusions from a probabilistic model. Anthropomorphizing for a moment, Monte Carlo methods can be interpreted as hallucinating possible worlds and evaluating those worlds according to how well they fit the model and how well they explain observations about the real world. From this perspective, the rejection sampling Monte Carlo method hallucinates complete random worlds, drawing conclusions only from those hallucinations that match the observed evidence. Likelihood weighting, another Monte Carlo method, hallucinates random worlds up to (but not including) the gathering of evidence, then weights any conclusions drawn from one of these hallucinated worlds by how well the world fits with the observations of the real world. Markov chain Monte Carlo also hallucinates possible worlds, but tries to be more systematic about it by continually adjusting its hallucination to try to better account for real world observations. There are a wide variety of abstract ideas from artificial intelligence that can be concretized as Monte Carlo inference. For example, streams and counterstreams [65] is an interesting proposal for modeling the interaction of top-down and bottom-up effects on visual perception. Unfortunately, the proposal is framed in terms of priming cognitive states, with no guidance provided on how such priming might be realized in a computational model.

In contrast, work on data-driven Markov chain Monte Carlo [64] has approached the same problem from a Monte Carlo inference perspective, resulting in a concrete model that seems to provide the most promising computational interpretation available of streams and counterstreams. Monte Carlo-based probabilistic inference holds the potential to systematize a wide range of artificial intelligence theories, offering to add both algorithmic alternatives and rational analysis to the existing intuitions.

This chapter surveys the mathematics of probabilistic inference, focusing on those aspects that will provide the foundation for the remainder of this thesis.

2.1 Probabilistic Models and Inference

In probabilistic models on discrete variables, $P(x = x_i)$ denotes the probability that the random variable x takes on the value x_i . This is often simply written as $P(x)$ or $P(x_i)$. Likewise, the joint probability of two variables is written $P(x = x_i, y = y_j)$ and indicates the probability that random variable x takes the value x_i and variable y takes the value y_j . Conditional probabilities are denoted $P(x = x_i | y = y_j)$ and indicate the probability that the random variable $x = x_i$, given that $y = y_j$. If y is the empty set, then $P(x|y) = P(x)$. Probability distributions on discrete variables satisfy several properties: $0 \leq P(x|y) \leq 1$, and if the set X contains all possible values for the variable x , then $\sum_{x_i \in X} P(x = x_i | y) = 1$.

For continuous variables, the terminology is slightly different. $P(x \in X)$ denotes the probability that the random variable x takes on a value in the set X . A probability density function $p(x = x_i)$ is then derived from this by the relation $\int_{x_i \in X} p(x = x_i) = P(x \in X)$. The terminology for joint and conditional distributions changes analogously. Probability distributions on continuous variables satisfy several properties: $0 \leq P(x \in X | y) \leq 1$, $p(x = x_i | y) \geq 0^1$, and if the set X contains all possible values for the variable x , then $\int_{x_i \in X} p(x = x_i | y) = 1$.

For the remainder of this thesis, references to distributions will be written as if

¹Note that for continuous variables, $p(\cdot)$ denotes the *density* of the probability distribution and is not restricted to be less than 1. For example, a uniform distribution on the interval $[0, \frac{1}{2}]$ has density 2 on that interval.

the distribution is over continuous variables. However, it should be understood that all methods are equally applicable to discrete variables unless otherwise noted.

There are a few useful rules for computing desired distributions from other distributions. First, marginal probabilities can be computed by integrating out a variable: $p(x|a) = \int_y p(x, y|a)$. Bayes' theorem declares that $p(x, y|a) = p(y|x, a)p(x|a)$, or equivalently,

$$p(y|x, a) = \frac{p(x, y|a)}{p(x|a)} = \frac{p(x|y, a)p(y|a)}{\int_y p(x|y, a)p(y|a) dy}$$

In problems of probabilistic inference, the following are specified: a set of variables \vec{x} ; a partition of the variables \vec{x} into three groups: \vec{e} (the evidence variables with observed values), \vec{q} (the query variables), and \vec{u} (the uninteresting variables); and a joint density $p(\vec{x}) = p(\vec{e}, \vec{q}, \vec{u})$ over those variables. The goal of inference, then, is to compute the distribution of the query variables given the observed evidence:

$$p(\vec{q}|\vec{e}) = \frac{p(\vec{q}, \vec{e})}{p(\vec{e})} = \frac{\int_{\vec{u}} p(\vec{e}, \vec{q}, \vec{u}) d\vec{u}}{p(\vec{e})} = \frac{\int_{\vec{u}} p(\vec{e}, \vec{q}, \vec{u}) d\vec{u}}{\int_{\vec{q}, \vec{u}} p(\vec{e}, \vec{q}, \vec{u}) d\vec{q} d\vec{u}}$$

Once the conditional distribution $p(\vec{q}|\vec{e})$ is in hand, it can be used answers queries² such as the expected value of some function $f(\vec{q})$:

$$\mathbb{E}_{p(\vec{q}|\vec{e})}[f(\vec{q})] = \int_{\vec{q}} f(\vec{q})p(\vec{q}|\vec{e}) d\vec{q}$$

For example, in a classification task, \vec{e} might be a set of observed object properties, \vec{q} might be the assignments of objects to classes, and \vec{u} might be the parameters governing the distribution of properties in each class. Suppose you were interested in whether two particular objects belonged to the same class. Letting $f(\vec{q})$ be an indicator function

$$f(\vec{q}) = \begin{cases} 1 & \text{if } \vec{q} \text{ assigns the two objects to the same class;} \\ 0 & \text{otherwise.} \end{cases}$$

then $\mathbb{E}_{p(\vec{q}|\vec{e})}[f(\vec{q})]$ would be the probability that the two objects belong to the same

class.

This is an elegant expression of the goals of inference, but unfortunately it is rarely possible to directly apply the inference formulae because the required integrals (or the analogous summations in the discrete case) are intractable for most probabilistic models. As a result, even after specifying the model and the inference task to be performed, it is still necessary to derive a method for performing that inference that does not require the evaluation of intractable integrals.

2.2 Approximate Inference

For this thesis, I focus on approximate probabilistic inference methods. While exact inference methods exist and are useful for certain classes of problems, exact inference in sophisticated models is generally intractable, because these methods typically depend on integrals, summations, or intermediate representations that grow unmanageably large as the state space grows large or even infinite.

There are two main classes of approximate inference: variational methods and Monte Carlo methods. *Variational methods* operate by first approximating the full model with a simpler model in which the inference questions are tractable. Next, the parameters of this simpler model are adjusted to minimize a measure of the dissimilarity between the original model and the simplified version; this adjustment is usually performed deterministically. Finally, the query is executed in the adjusted, simplified model.

In contrast, *Monte Carlo* methods draw a set of samples from the target dis-

²Another popular inference goal is to find the maximum *a posteriori* (MAP) value of the query variables: $\vec{q}_{MAP} = \arg \max_{\vec{q}} p(\vec{q}|e)$. MAP values are typically used to find the “best explanation” of a set of data. Unfortunately, they do not satisfy intuitive consistency properties. In particular, a change-of-variables transformation of the target distribution is likely to change the MAP value. More concretely: let $f : q \rightarrow q^*$ be some invertible function, let $F : Q \rightarrow Q^*$ be the analogous set-valued invertible function $F(Q) = \{f(q)|q \in Q\}$. Consider the change-of-variables transformed distribution $P^*(Q^*|e) = P(F^{-1}(Q^*)|e)$ with density $p^*(q^*|e) = \frac{d}{dq^*} P^*(Q^*|e)$. Intuitive consistency is violated because, in general, $\arg \max_q p(q|e) \neq f^{-1}(\arg \max_{q^*} p^*(q^*|e))$, implying that the choice of representation can change the “best explanation.” This and other shortcomings notwithstanding, MAP values can also be estimated using Monte Carlo methods.

tribution; inference questions are then answered by using this set of samples as an approximation of the target distribution itself.

Variational methods have the advantage of being deterministic; the corresponding results, however, are in the form of a lower bound on the actual desired quantity, and the tightness of this bound depends on the degree to which the simplified distribution can model the target distribution. Furthermore, standard approaches to variational inference such as variational message passing [66] restrict the class of models to graphical models in which adjacent distributions are conjugate³. For example, when conjugacy assumptions are not satisfied, [66] recommends reverting to a Monte Carlo approach. In contrast, Monte Carlo techniques are applicable to all classes of probabilistic models. They are also guaranteed to converge – if you want a more accurate answer, you just need to run the inference for longer; in the limit of running the Monte Carlo algorithm forever, the sampled approximation converges to the target distribution. Furthermore, it is possible to construct hybrid inference algorithms in which variational inference is used for some parts of the model, while Monte Carlo methods are used for the rest. These mixed approaches, however, are outside the scope of this work. For this thesis, I concentrate on Monte Carlo methods because the mathematics for this class of inference supports inference composition in a way that parallels model composition (For example, see the description of hybrid kernels in section 2.4). Notwithstanding the particular focus of this thesis, the stochastic automata developed here (BLAISE SDK graphs, chapter 3) could be used to model processes other than Monte Carlo inference, including other inference techniques (e.g. belief propagation [49, 31]) and even non-inferential processes.

³A prior distribution $p(\theta)$ is said to be *conjugate* to a likelihood $p(x|\theta)$ if the posterior distribution $p(\theta|x)$ is of the same functional form as the prior. Conjugacy is generally important because it allows key integrals to be computed analytically, and because it allows certain inference results to be represented compactly (as the parameters of the posterior distribution).

2.3 Monte Carlo Methods

There are a wide variety of Monte Carlo methods, but they all share a common recipe. First, draw a number of samples $\langle \vec{q}_1, \vec{u}_1 \rangle, \dots, \langle \vec{q}_N, \vec{u}_N \rangle$ from the distribution $p(\vec{q}, \vec{u}|\vec{e})$, and then approximate the interest distribution using

$$p(\vec{q}|\vec{e}) \approx \tilde{p}_N(\vec{q}|\vec{e}) = \frac{1}{N} \sum_{i=1}^N \delta_{\vec{q}_i}(\vec{q})$$

where $\delta_{\vec{q}_i}$ is the Dirac delta function⁴. As the number of samples increases, the approximation (almost surely) converges to the true distribution: $\tilde{p}_N(\vec{q}|\vec{e}) \xrightarrow[N \rightarrow \infty]{a.s.} p(\vec{q}|\vec{e})$. Expectations can similarly be approximated from the Monte Carlo samples:

$$\mathbb{E}_{p(\vec{q}|\vec{e})}[f(\vec{q})] = \int_{\vec{q}} f(\vec{q}) p(\vec{q}|\vec{e}) d\vec{q} \approx \int_{\vec{q}} f(\vec{q}) \frac{1}{N} \sum_{i=1}^N \delta_{\vec{q}_i}(\vec{q}) d\vec{q} = \frac{1}{N} \sum_{i=1}^N f(\vec{q}_i)$$

If it were generally easy to draw samples directly from $p(\vec{q}, \vec{u}|\vec{e})$, the Monte Carlo story would end here. Unfortunately, this is typically intractable due to the same integrals that made it intractable to compute $p(\vec{q}|\vec{e})$ exactly. Fortunately, a range of techniques have been developed for producing samples from $p(\vec{q}, \vec{u}|\vec{e})$ indirectly.

One of the simplest Monte Carlo techniques is rejection sampling. In rejection sampling, samples from the conditional distribution $p(\vec{q}, \vec{u}|\vec{e})$ are produced by generating samples from the joint distribution $p(\vec{q}, \vec{u}, \vec{e})$ and discarding any samples that disagree with the observed evidence values. Rejection sampling is extremely inefficient if the observed evidence is unlikely under the joint distribution, because almost all of the samples will disagree with the observed evidence and be discarded. Furthermore, as the amount of observed data increases, any particular set of observations gets increasingly unlikely.

Importance sampling is a Monte Carlo technique that avoids discarding samples by only sampling values for \vec{q} and \vec{u} ; these samples are then weighted by how well

⁴The Dirac delta function $\delta_{x_0}(x)$ has the properties that it is non-zero only at $x = x_0$, $\int_X \delta_{x_0}(x) dx = 1$, and $\int_X f(x) \delta_{x_0}(x) dx = f(x_0)$. The Dirac delta can be thought of as the derivative of the Heaviside step function $H_{x_0}(x) = \begin{cases} 0 & \text{for } x < x_0; \\ 1 & \text{for } x \geq x_0. \end{cases}$

they conform to the evidence. More specifically, samples $\langle \vec{q}_i, \vec{u}_i \rangle$ are drawn from a proposal distribution $q(\vec{q}, \vec{u})$ and assigned weights $w(\vec{q}_i, \vec{u}_i) = \frac{p(\vec{q}_i, \vec{u}_i, \vec{e})}{q(\vec{q}_i, \vec{u}_i)}$. Observing that $p(\vec{q}_i, \vec{u}_i, \vec{e}) = q(\vec{q}_i, \vec{u}_i)w(\vec{q}_i, \vec{u}_i)$, the Monte Carlo approximation of the conditional distribution becomes

$$p(\vec{q}|\vec{e}) \approx \frac{\sum_{i=1}^N \delta_{\vec{q}_i}(\vec{q})w(\vec{q}_i, \vec{u}_i)}{\sum_{i=1}^N w(\vec{q}_i, \vec{u}_i)}$$

Efficient importance sampling requires that the proposal distribution $q(\vec{q}, \vec{u})$ produces samples from the high-probability regions of $p(\vec{q}, \vec{u}|\vec{e})$. Choosing a good proposal distribution can be very difficult, becoming nearly impossible as the dimensionality of the search space increases.

2.4 Markov Chain Monte Carlo

Markov chain Monte Carlo (MCMC) is an inference method designed to spend most of the computational effort producing samples from the high probability regions of $p(\vec{q}, \vec{u}|\vec{e})$. In this method, a stochastic walk is taken through the state space $\vec{q} \times \vec{u}$ such that the probability of being in a particular state $\langle \vec{q}_i, \vec{u}_i \rangle$ at any point in the walk is $p(\vec{q}_i, \vec{u}_i|\vec{e})$. Therefore, samples from $p(\vec{q}, \vec{u}|\vec{e})$ can be produced by recording the states visited by the stochastic walk. The stochastic walk is a Markov chain: the choice of state at time $t + 1$ depends only on the state at time t . Formally, if $s_t \in \vec{q} \times \vec{u}$ is the state of the chain at time t , then $p(s_{t+1}|s_1, \dots, s_t) = p(s_{t+1}|s_t)$. Because Markov chains are history-free, they can be run for an unlimited number of iterations without consuming additional memory space; contrast this with classic backtracking search strategies which maintain a complete history of visited states and a schedule of states to be visited. The history-free property also means that the MCMC stochastic walk can be completely characterized by $p(s_{t+1}|s_t)$, known as the transition kernel. I will use the notation $K(s_t \rightarrow s_{t+1}) = p(s_{t+1}|s_t)$ for transition kernels to emphasize the directionality of the movement through the state space. The transition kernel K is a linear transform, such that if $p_t = p_t(s)$ is the row vector encoding the probability of the walk being in state s at time t , then $p_{t+1} = p_t K$.

If the stochastic walk starts from state s_0 , such that the distribution over this initial state is the delta distribution $p_0 = \delta_{s_0}(s)$, then the state distribution for the chain after step t is $p_t = p_0 K^t$. The key to Markov chain Monte Carlo is to choose K such that $\lim_{t \rightarrow \infty} p_t = p(\vec{q}, \vec{u} | \vec{e})$, regardless of choice of s_0 ; kernels with this property are said to converge to an equilibrium distribution $p_{eq} = p(\vec{q}, \vec{u} | \vec{e})$. Convergence is guaranteed if both:

- p_{eq} is an invariant (or stationary) distribution for K . A distribution p_{inv} is an invariant distribution for K if $p_{inv} = p_{inv} K$.
- K is *ergodic*. A kernel is ergodic if it is *irreducible* (any state can be reached from any other state) and *aperiodic* (the stochastic walk never gets stuck in cycles).

Markov chain Monte Carlo can be viewed as fixed point iteration on the domain of probability distributions, where K is the iterated function and p_{eq} is the unique fixed point, even though in practice K is iteratively applied to a *sample* from p_{eq} , rather than p_{eq} itself, side-stepping the issue of explicitly representing distributions.

Transition kernels compose well. Let K_1 and K_2 be two transition kernels with invariant distribution p_{inv} . The *cycle hybrid kernel* $K_{cycle} = K_1 K_2$ is the result of first taking a step with K_1 , then taking a step with K_2 . K_{cycle} has the same invariant distribution p_{inv} . Kernels can also be composed using a *mixture hybrid kernel* $K_{mixture} = \alpha K_1 + (1 - \alpha) K_2$ for $0 \leq \alpha \leq 1$, which is the result of stochastically choosing to apply either K_1 or K_2 , with α being the probability of choosing K_1 . Mixture hybrid kernels also maintain the invariant distribution p_{inv} . These hybrid kernels do not guarantee ergodicity, but it is generally very easy to show that the composite kernel is ergodic. Hybrid kernels are the key that will enable MCMC-based inference to compose in the same way that probabilistic models compose.

Kernel composition is only useful to the extent that effective base kernels can be generated. The most common recipe for constructing an MCMC transition kernel with a specific equilibrium distribution is the Metropolis-Hastings method [43, 26], which converts an arbitrary proposal kernel $q(s_t \rightarrow s_*)$ into a transition kernel

with the desired invariant distribution $p_{eq}(s)$. In order to produce a sample from a Metropolis-Hastings transition kernel, one first draws a proposal $s_* \sim q(s_t \rightarrow s_*)$, then evaluates the Metropolis-Hastings acceptance probability

$$A(s_t \rightarrow s_*) = \min \left(1, \frac{p(s_*)q(s_* \rightarrow s_t)}{p(s_t)q(s_t \rightarrow s_*)} \right).$$

With probability $A(s_t \rightarrow s_*)$ the proposal is accepted and $s_{t+1} = s_*$; otherwise the proposal is rejected and $s_{t+1} = s_t$. Intuitively, Metropolis-Hastings kernels tend to accept moves that lead to higher probability parts of the state space due to the $\frac{p(s_*)}{p(s_t)}$ term, while also tending to accept moves that are easy to undo due to the $\frac{q(s_* \rightarrow s_t)}{q(s_t \rightarrow s_*)}$ term. Because Metropolis-Hastings kernels only evaluate $p(s)$ as part of the ratio $\frac{p(s_*)}{p(s_t)}$, one may include in every evaluation of $p(s)$ an unknown normalizing constant without altering the kernel's transition probabilities. For inference, this means that instead of computing the generally intractable integral involved in evaluating $p(s_t) = p(\vec{q}_t, \vec{u}_t | \vec{e}) = \frac{p(\vec{q}_t, \vec{u}_t, \vec{e})}{p(\vec{e})} = \frac{p(\vec{q}_t, \vec{u}_t, \vec{e})}{\int_{\vec{q}, \vec{u}} p(\vec{q}, \vec{u}, \vec{e})}$, we can let $c = \frac{1}{p(\vec{e})}$ be an unknown normalizing constant and evaluate $p(s_t) \propto p(\vec{q}_t, \vec{u}_t, \vec{e})$.

2.5 Transdimensional MCMC

Sophisticated models often have an unknown number of variables. For example, a mixture model typically has parameters associated with each mixture component; if the number of components is itself to be inferred, the model has an unknown number of variables. MCMC kernels that change the parameterization of the model, such as those that change the dimensionality of the parameter space, must ensure that the reparameterization is accounted for⁵.

Reversible jump MCMC [21], also known as the Metropolis-Hastings-Green method,

⁵For example, consider a Kernel that can reparameterize a model from a variable x distributed uniformly on the range $[0, 1]$ to a variable x' distributed uniformly on $[0, \frac{1}{2}]$. Even though there is a one-to-one correspondence between x and x' values, e.g. $x = 2x'$, the two parameterizations have different densities. Specifically, for any value of $x \in [0, 1]$, the probability density is 1, whereas for any value of $x' \in [0, \frac{1}{2}]$ the probability density is 2. Any “compressing” or “stretching” of the state space must be accounted for. As described in the context of Reversible jump MCMC in this section, the Jacobian of the transformation is the mathematical tool for measuring this state space distortion.

is an extension of the Metropolis-Hastings method for transdimensional inference. In the Reversible Jump framework, sampling from the proposal distribution $q(s_t \rightarrow s_*)$ is broken into two phases. First, a vector of random variables v is sampled from a distribution $q(v)$; note that this distribution is not conditioned on s_t . Then s_* is computed from s_t and v using an invertible deterministic function g ; that is, $\langle s_*, v' \rangle = g(\langle s_t, v \rangle)$ and $\langle s_t, v \rangle = g^{-1}(\langle s_*, v' \rangle)$, where the dimensionality of $\langle s_t, v \rangle$ matches the dimensionality of $\langle s_*, v' \rangle$. Finally, the Metropolis-Hastings acceptance ratio is adjusted to reflect any changes in the parameter space caused by this move using a Jacobian factor:

$$A(s_t \rightarrow s_*) = \min \left(1, \frac{p(s_*)q(v')}{p(s_t)q(v)} \left| \frac{\partial \langle s_*, v' \rangle}{\partial \langle s_t, v \rangle} \right| \right)$$

where $\left| \frac{\partial \langle s_*, v' \rangle}{\partial \langle s_t, v \rangle} \right|$ is the Jacobian factor: the absolute value of the determinant of the matrix of first-order partial derivatives of the function g .

2.6 Tempered Inference

A number of Monte Carlo inference variants, including simulated annealing and parallel tempering, operate by changing the “temperature” τ of the interest distribution:

$$p_{\text{tempered}}(s) \propto p(s)^{1/\tau}$$

where $p_{\text{tempered}}(s)$ reduces to $p(s)$ when $\tau = 1$. As τ goes to 0, $p_{\text{tempered}}(s)$ concentrates all of its mass on its modes; therefore, sampling from $p_{\text{tempered}}(s)$ for very small τ is much more likely to produce the maximum *a posteriori* value than sampling from $p(s)$. However, such “peaky” interest distributions, whether they arise naturally or through tempering, are generally more difficult for Monte Carlo methods to handle effectively. For example, Metropolis-Hastings kernels operating on a “peaky” distribution are much more likely to have their proposals rejected.

In contrast, as τ goes to ∞ , $p_{\text{tempered}}(s)$ gets increasingly flat, and Monte Carlo methods can produce samples very easily. The disadvantage of these high- τ samples is that they are less likely to come from high-probability regions of the original interest

distribution $p(s)$. Simulated annealing and parallel tempering both use a sequence of tempered distributions from $\tau = 1$ to a τ large enough to make inference easy; the intuition is to leverage results from the easy-inference values of τ to perform better on the harder τ values.

In simulated annealing [17, 32], τ is initialized to a large value. Then, as MCMC proceeds, τ is gradually decreased. The hope is that the stochastic walk will find the mode of the distribution while τ is large, and will settle in that mode as τ is decreased. Once τ is small, the stochastic walk is unlikely to leave that mode (assuming a “peaky” distribution); therefore, simulated annealing is most useful for locating the maximum *a posteriori* value.

In parallel tempering [18, 15], multiple MCMC chains are run in parallel, each at a different fixed value of τ . Occasionally, swaps of adjacent chains are proposed and evaluated according to the Metropolis-Hastings acceptance ratio. Samples are only collected from the lowest τ chain. One interpretation of parallel tempering is that the high τ chains act as proposal distributions for the lower τ chains. The samples gathered from parallel tempering should be representative of the whole interest distribution (as opposed to simulated annealing, which produces samples only from one mode). Parallel tempering provides a generic means for the MCMC inference to move efficiently between modes of the interest distribution, even when those modes are widely separated by low-probability regions. Without parallel tempering, the probability that plain MCMC will make these moves becomes vanishingly small unless a great deal of problem specific knowledge is used to construct clever proposal distributions.

2.7 Particle Filtering

Particle filtering, also known as Sequential Monte Carlo, is a population-based Monte Carlo method similar to importance sampling. It is typically applied to dynamic models with unobserved variables x_i forming a Markov chain such that $p(x_i|x_0, \dots, x_{i-1}) = p(x_i|x_{i-1})$ and with observed variables y_i modeled by $p(y_i|x_i)$, yielding a joint density

$p(x_0, \dots, x_n, y_1, \dots, y_n) = p(x_0) \prod_{i=1}^n p(x_i|x_{i-1})p(y_i|x_i)$. Inference by particle filtering produces an approximation to $p(x_n|y_1, \dots, y_n)$.

This technique unrolls inference over the same timeline used to index the dynamic model, such that the inference results for $p(x_n|y_1, \dots, y_n)$ together with the observation y_{n+1} are all that is needed to infer $p(x_{n+1}|y_1, \dots, y_{n+1})$. Inference is achieved using a population of “particles”: weighted samples $\langle x_i^j, w_i^j \rangle$ which together form a Monte Carlo estimate of $p(x_n|y_1, \dots, y_n) \approx \sum_j \delta_{x_i^j}(x_i)w_i^j$ (the weights are normalized such that $\sum_j w_i^j = 1$).

Inference is initialized by drawing a number of particles from the prior distribution on states and assigning each particle an equal weight:

$$x_0^j \sim p(x_0); \quad w_0^j = \frac{1}{\#particles}$$

Inference is then advanced to the next time step by stochastically advancing each particle according to an importance distribution q , such that $x_i^j \sim q(x_i^j|x_{i-1}^j, y_i)$. The simplest cases are those in which it is tractable to sample from $q(x_i^j|x_{i-1}^j, y_i) = p(x_i^j|x_{i-1}^j)p(y_i|x_i^j)$. Otherwise, a common choice for q is $q(x_i^j|x_{i-1}^j, y_i) = p(x_i^j|x_{i-1}^j)$, though any approximation can be used. Weights are then updated as in importance sampling:

$$w_i^j = \frac{p(x_0^j, \dots, x_i^j, y_1, \dots, y_i)}{\prod_{k=1}^i q(x_k^j|x_{k-1}^j, y_k)} = w_{i-1}^j \frac{p(x_i^j|x_{i-1}^j)p(y_i|x_i^j)}{q(x_i^j|x_{i-1}^j, y_i)}$$

Next, the weights are renormalized to sum to 1:

$$w_i^j \leftarrow \frac{w_i^j}{\sum_{k=1}^{\#particles} w_i^k}$$

Finally, the particles may be resampled and the weights set to equal values:

$$x_i'^j \sim p(x_i'^j = x_k^j) = w_k^j; \quad w_i'^j = \frac{1}{\#particles}$$

Without resampling, most of the particle weights would drift towards 0; resampling effectively kills off particles with small weights while duplicating particles with large

weights. Resampling is often only performed when certain criteria are met, such as when an estimate of the number of effective particles (i.e., particles with relatively large weight) falls below a predetermined threshold.

Chapter 3

The BLAISE State–Density–Kernel Graphical Modeling Language

My thesis is that a framework for probabilistic inference can be designed that enables efficient composition of both models and inference procedures, that is suited to the representational needs of emerging classes of probabilistic models, and that supports recent advances in inference.

In this chapter, I support this thesis by introducing the BLAISE State–Density–Kernel graphical modeling language and showing how this language supports composition of models and inference procedures.

By the end of this chapter, you will understand all the elements of the BLAISE SDK graphical modeling language. You will be able to draw complete graphical models, including graphical representations of inference, for sophisticated models such as multi-feature non-parametric mixture models, and you will understand how models can be built up iteratively by composing existing probabilistic models and inference methods with minimal effort. This chapter also provides the foundation for chapters 4–6, which will discuss transformations of SDK models, a virtual machine that can execute SDK models, and applications built using BLAISE.

This chapter introduces a graphical modeling language, including several symbols. Each symbol is described as it is introduced. For reference, appendix B also supplies a

complete legend of symbols, including the page on which the symbol was introduced.

3.1 An overview of the BLAISE modeling language

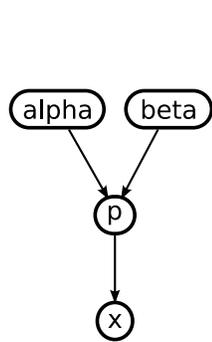
In order to fully specify a probabilistic modeling application, three things must be described. One thing the modeler must describe is the state space. This is a description of the domain of the problem: what are the variables we might be interested in? What values can those variables take on? Could there be an unknown number of variables (for example, could there be an unknown number of objects in the world we are trying to describe, such as an unknown number of airplanes in an aircraft tracking problem?) Are there structural constraints amongst the variables (for example, is every variable of type A associated with a variable of type B?) One of the BLAISE modeling language’s three central abstractions, **State**, is devoted to expressing these aspects of the model.

The state space typically describes a vast number of possible variable instantiations, most of which the modeler is not very interested in. The second central abstraction, **Density**, allows the modeler to describe how interesting a particular state configuration is. For discrete probabilistic models, this is typically the joint probability mass function. If continuous variables are used, then **Density** would represent the joint probability density function (from which the abstraction derives its name). When describing how to score a **State**, the modeler will be expressing things such as: how does the joint score decompose into common pieces, such as standard probability distributions? How does the score accommodate state spaces with unknown numbers of objects – does it have patterns that repeat for each one of these objects? The **Density** abstraction is designed to represent the modeler’s answers to these questions.

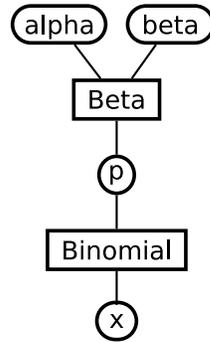
With **State** and **Density** in hand, the modeler can now express models, but cannot yet say how to extract information from these models. As described in chapter 2, there are a wide variety of inference techniques that can be applied. BLAISE focuses on those inference techniques that can be described as history-free stochastic walks

$$p(x|p, \alpha, \beta) = \text{Beta}(p|\alpha, \beta) \cdot \text{Binomial}(x|p)$$

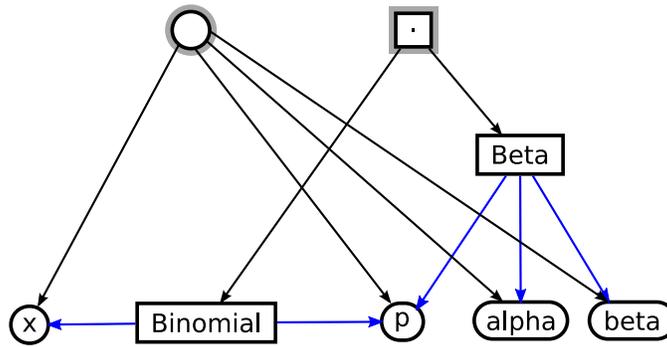
(a) Joint Density Equation



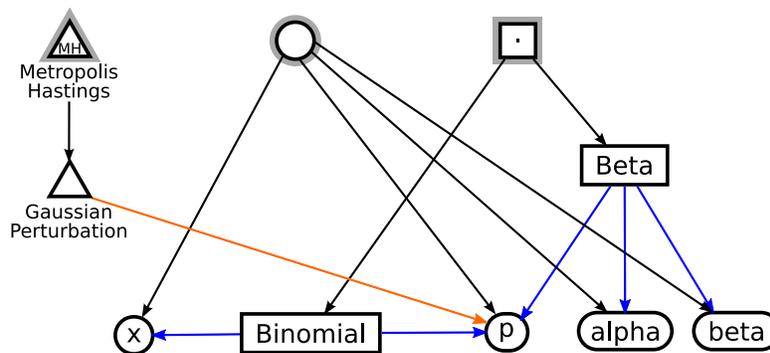
(b) Bayes Net



(c) Factor Graph



(d) BLAISE, without inference



(e) BLAISE, with inference

Figure 3-1: A preview of the BLAISE modeling language, showing the same simple Beta-Binomial model as (a) a joint probability density equation, (b) a Bayes net, (c) a factor graph, (d) a BLAISE probabilistic model (no inference), and (e) a BLAISE probabilistic model with inference.

through a State space, guided by the Density¹. All such walks can be completely described by a transition kernel: an expression of the probability that the stochastic walk will make a particular step in the state space, given the state the walk is currently at. To describe a transition kernel, a modeler will have to make choices such as: which variables in the state space will change on this step? How exactly will these variables be updated – are there common update procedures that can be used? How will these update rules be coordinated so that the whole state space is explored efficiently – that is, how are fragments of an inference algorithm composed? How does the inference method accommodate state spaces with unknown numbers of objects? Often the modeler will want to maintain a certain relationship between the Density and the exploration of the state space; for example, a modeler designing an Markov chain Monte Carlo-based inference method will want to ensure that the transition kernel converges to the Density as an invariant distribution. How will the modeler meet this goal? These considerations are the focus of the **Kernel** abstraction in BLAISE.

A common design tenet runs throughout the entire modeling language: support composability. That is, it should be easy for the modeler to reuse existing models in the creation of new models. For example, if the modeler has constructed a State–Density–Kernel representation of a Chinese Restaurant Process, it should be easy for the modeler to reuse this representation to create a CRP-based mixture model. In most cases, in fact, the SDK for the original model should not need to be modified at all – even the same inference procedure should continue to work in the new model, despite the fact that there are now other States in the state space and other Densities affecting the joint probability density. Realizing this design philosophy will mean that if a modeler extends an existing model or composes several existing models, development resources can be reserved for the truly novel parts of the new model. It is my hypothesis that such an approach will provide the leverage required to effectively engineer sophisticated models of increasing complexity, such as are becoming ever more important in artificial intelligence, cognitive science, and commercial applications.

¹The history-free limitation is restrictive, because history-dependent stochastic walks can also be modeled by augmenting the State space with an explicit representation of the history.

This chapter will compare compare and contrast the BLAISE modeling language with classical graphical modeling languages such as Bayes nets and factor graphs. It should be noted that BLAISE models are strictly more expressive than factor graphs; see section 6.5 for a simple demonstration of how any factor graph can be translated to a BLAISE model.

Although BLAISE SDK graphs are presented here in the specific context of Monte Carlo methods for probabilistic inference, the SDK foundation (consisting of a domain described using States, functions over the domain described using Densities, and a stochastic process for domain exploration described using Kernels, together with simple composition and locality rules for each of these representations) can also serve as a general framework for expressing and manipulating any stochastic (or deterministic) automaton.

3.2 BLAISE States

The state space describes the domain of the inference problem; that is, the variables and their valid settings. All probabilistic modeling languages have some representation of the state space: graphical modeling languages, such as Bayes nets and factor graphs, use nodes to represent variables (as in figure 3-2), whereas programmatic modeling languages, such as BLOG [45], allow the user to declare variables. BLAISE follows in the graphical modeling tradition by representing variables as graph nodes called *States*. State nodes are also typed, carrying information about what values the represented variable can take on. For example, a State node might be typed as a continuous variable, indicating that it will take real numbers as values.

Unlike classical graphical modeling languages, however, BLAISE requires that its State nodes be organized into a single rooted tree via containment (has-a) links in the graph (See figure 3-3). This organization is the foundation of State composition in BLAISE – it allows the modeler to take several States and bundle them together as children of some parent State. Note that the space of States is closed under this composition structure: composing several States produces another State.

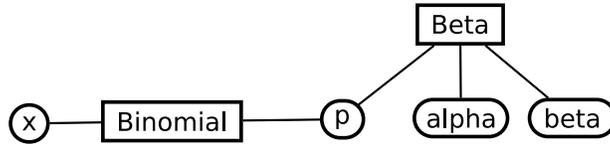


Figure 3-2: A simple graphical model for a single draw x from a beta-binomial model, drawn as a factor graph. Several of the examples in this chapter build on this familiar model, though most will ignore the conjugacy properties of the model. Exploiting conjugacy in BLAISE will be discussed in section 4.6.

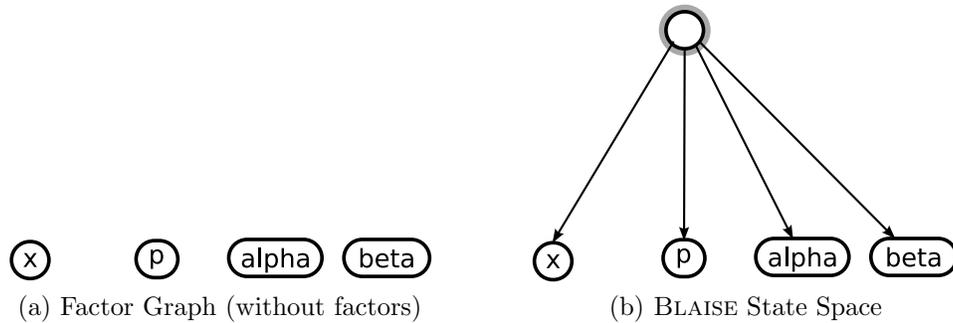


Figure 3-3: Omitting the factors from the beta-binomial model factor graph in figure 3-2 leaves just the variables, representing the state space of the model, as in (a). Figure (b) shows the same state space as it might be implemented in BLAISE. States in BLAISE models form trees. The State in the BLAISE model that does not have an analog in the factor graph (i.e., the root of the tree) is used to compose diverse States into a single state space. The root State is highlighted with a gray annulus.

The tree-structured organization of States is a critical enabler in modeling repeated structure in the state space. Information about repeated structure is commonly represented in a graphical modeling language using “plate notation” – drawing a box (a “plate”) containing the variables that will be repeated, and writing a number in the corner of the box to indicate how many times that structure will be repeated. Plate notation has several limitations. Most significantly, state spaces represented using plate notation are not closed under composition: composing several variable nodes produces a new class of object (a plate) rather than a variable. This in turn means that the number of copies of a plate is *not* part of the state space. This information is not available as a variable, so, for example, one cannot express a prior over the number of copies of a plate nor perform inference to determine how many copies of the plate should be used. This prevents an intuitive expression of even simple models such as mixture models, if the number of components is not known a

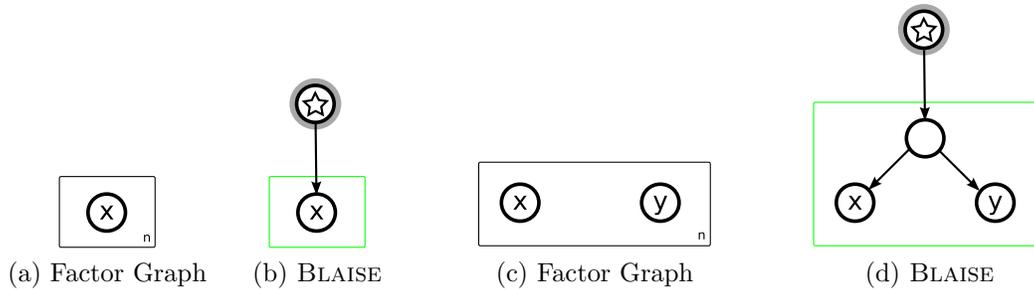


Figure 3-4: Figures (a) and (c) show the state spaces for simple factor graphs, using plate notation to represent repetition. Figures (b) and (d) show the corresponding BLAISE state spaces. States marked with a star are BLAISE Collection States. The unmarked State in (d) is a generic composite State containing x and y .

priori. Expressing non-parametric mixture models is even more complicated.

There are a number of other important shortcomings of plate notation. Plate notation is most often used in the context of Bayes nets, where there is the additional limitation that the notation does not express how the model’s joint density should factor across the plate boundary. Inference procedures also need to account for repeated structures in the state space, particularly when the number of repetitions is not fixed *a priori*. Finally, plate notation only allows plates to interact by having one plate embedded in another; it does not permit plates to intersect, nor interact in other more complex relationships, without making the meaning ambiguous. This makes it challenging to express many interesting models. Each of these limitations will be addressed in this chapter (specifically in sections 3.5, 3.6 and 3.7).

Instead of plates, BLAISE uses State composition to capture repeated structure. BLAISE allows States to have arbitrarily-sized collections of children. Such Collection States are used to capture the idea of repetition. For example, a model that would be denoted in plate notation as a single variable x inside a plate would be captured in BLAISE as a Collection State with a collection of x States as children (see Figure 3-4 (a) and (b)). Composition allows the same containment mechanism to be used for repeated state structure rather than just single states. For example, a model that would be denoted in plate notation as two variables x and y inside a plate would be captured in BLAISE as a Collection State with a collection of composite States, where each composite has an x and a y (see Figure 3-4 (c) and (d)). For easy interpretation,

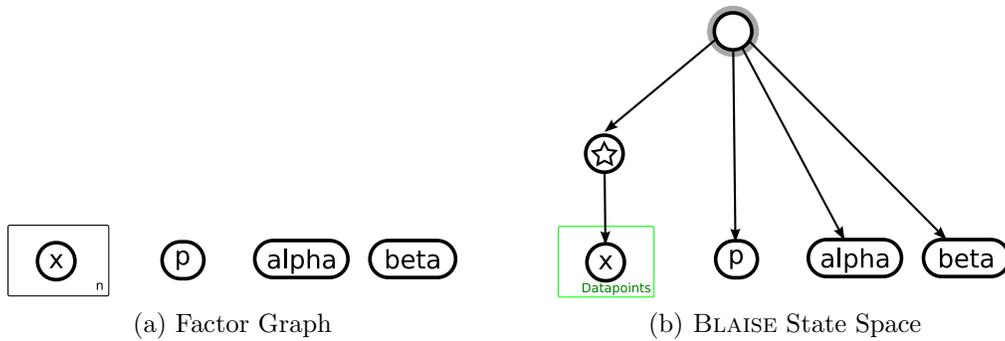


Figure 3-5: The beta-binomial models from Figure 3-3 can be extended to model multiple datapoints drawn from the same binomial distribution. This figure shows state space of this extended model, in plated factor graph notation and as a BLAISE State structure.

BLAISE will also include plate-like boxes surrounding the repeated structure. However it must be emphasized that these ornamentations carry no new information – they simply highlight the children of a Collection State, allowing the grouping to be seen at a glance, much as a syntax highlighting text editor might highlight balanced pairs of parentheses without providing any additional information.

Reifying the repetition of State structure using Collection States remedies the weakness of plate notation wherein the number of copies of a repeated structure is not available as a variable. Because the Collection State is a State like any other, it serves as a variable in the State space. Thus the computation of the joint density can naturally reference the size of the Collection State (the representation of the joint density will be described shortly in section 3.3).

In order to perform Monte Carlo inference in state spaces with repeated structure where the repetition count is not known *a priori*, it will be necessary to consider states with different repetition counts. That is, it will be necessary, at inference time, to allow instances of the repeated structure to be added to and removed from the state space. Thus, the topology of BLAISE States is considered to be mutable at inference time, so that children may be added and removed from Collection States. It also follows that the State *topology* carries information. Consider, as an example, the information contained in the size of a Collection State, which might be used to compute the joint density (as above), or might itself be the target of inference (e.g. for

a query such as “how many mixture components are required to explain this data?”).

An interesting effect of allowing the State topology to bear information is that many models that would normally require the use of integer indices no longer require such indices. For example, consider a mixture model, where the number of mixture components is fixed *a priori*, but where the assignment of data to components is to be inferred. A Bayes net for such a model would assign a unique integer index from the range $[0, \text{number of components} - 1]$ to each component (a name, in essence), and each data point would have associated with it a component index from the same range (see figures 3-6a and 3-6b). Inference is then a matter of choosing appropriate values for the integer indices associated with the datapoints. In a BLAISE model, it would be more natural to use a Collection State to represent each component, with the data points currently assigned to each component being the children of that component’s Collection State (see figures 3-6c and 3-6d). Inference is then a matter of moving data points from one component to another (figure 3-6e). This formulation has several advantages. First, it is more parsimonious insofar as the components of a mixture model usually do not actually have an order; the component indices in the Bayes net formulation are an artifact of the formalism that must be explicitly worked around when it comes time to compute the joint density or to evaluate a state in order to answer a query. Second, integer indices are often assumed to be contiguous, which imposes several inefficiencies in the implementation of the system. For example, deleting a component with a mid-valued component index will require changing the component index of at least one other component, otherwise the existing components will not have contiguous indices. Changing the value of a component index is inefficient because it requires finding all the data points associated with the component and updating their component index as well; thus, deleting a component in an integer-indexed model is usually implemented as an operation with time cost linear in the number of datapoints rather than the constant-time operation it is in a BLAISE model in which data point assignment is represented directly by the State topology.

Because BLAISE allows the topology of the State space to bear information, there

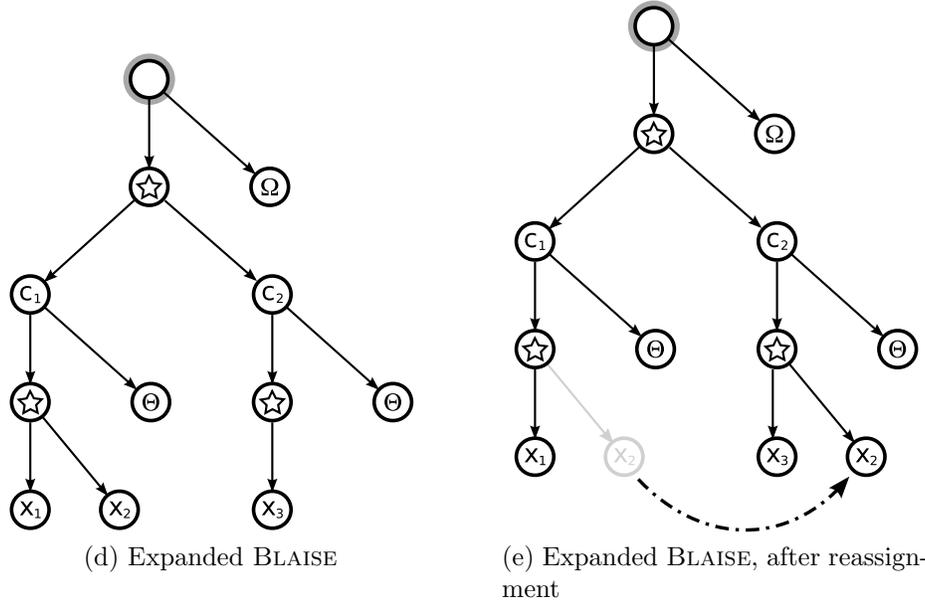
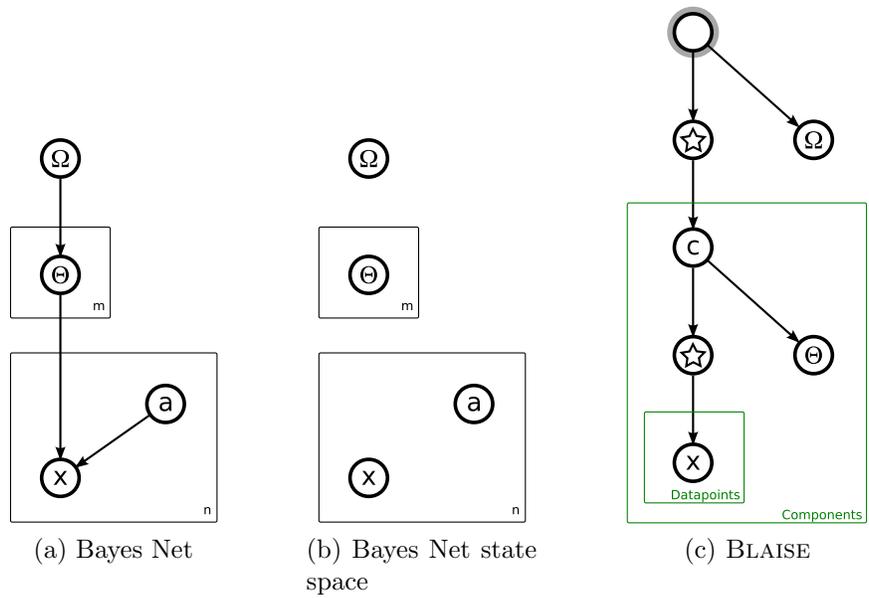


Figure 3-6: Mixture models are among the simplest models with interesting repeated structure. (a) shows a simple mixture model, represented as a Bayes net. There are m components, with Θ representing the parameters for a component. All the Θ variables are governed by a common hyperparameter Ω . There are also n datapoints, where the value of the datapoint is x , and $a \in [0, n - 1]$ encodes the component to which the associated datapoint is assigned. (b) shows just the state space for this Bayes net. (c) shows the the state space for a mixture model in BLAISE notation. Rather than using integer-valued component assignment variables (a in the Bayes nets), the BLAISE model uses Collection States for each component, where each Collection State contains just those datapoints assigned to the component. (d) shows an expanded version of this BLAISE model with two components and three datapoints, and (e) demonstrates how the State structure would change when datapoint x_2 is moved from component c_1 to component c_2 .

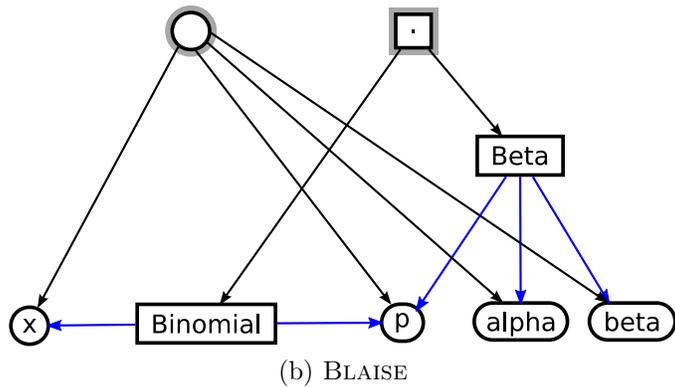
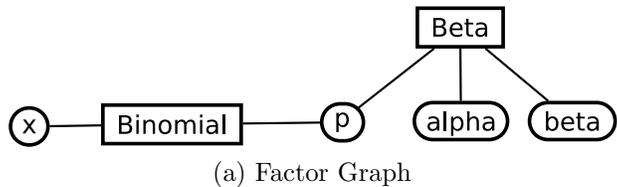
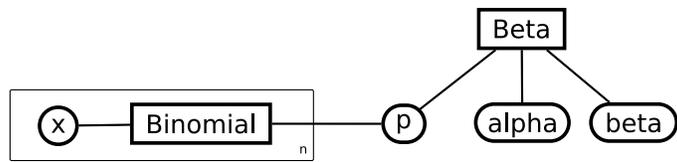


Figure 3-7: A model for a single draw x from a beta-binomial model, drawn as (a) a factor graph, and (b) a BLAISE model. In BLAISE models, Densities form trees. Densities also have States as children, encoding the portions of the State hierarchy that will be used when evaluating the Density. Note that the Density \rightarrow State connections reflect the factor \rightarrow node connections in the graphical model. The Density without a graphical model analog represents the (multiplicative) composition of individual Densities into a Density over the whole state space. The gray annulus around this Density highlights it as the root Density.

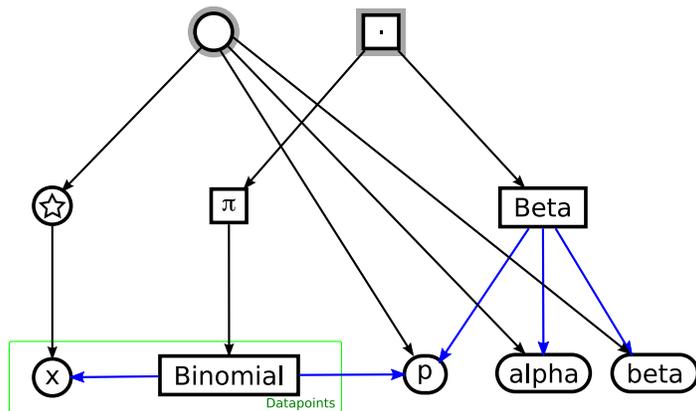
may be cases in which a reference to more than one State is required. For example, in an admixture model, a State may belong to more than one mixture component simultaneously. To capture this type of pattern, BLAISE States support *state-to-state dependency* links in addition to has-a links. These links are permitted to connect the States in non-tree-structured ways. State-to-state dependency links are also used to model constraints in the State space, as described in section 3.7.

3.3 BLAISE Densities

Whereas States are used to model the domain of the state space for a probabilistic model, *Densities* are used to describe the joint probability density function over that state space. It is often advantageous to decompose the joint probability density



(a) Graphical Model



(b) BLAISE Model

Figure 3-8: A model for a multiple draws from a beta-binomial model, drawn as (a) a factor graph, and (b) a BLAISE model. The density labeled π is a Multiplicative Collection Density; it composes any number of child Densities by computing the product of their values.

function into a number of simpler Densities that only depend on a subset of the state space variables (i.e., a projection of the state space onto a lower-dimensional subspace). For example, Bayes nets decompose the joint Density into a product of conditional probabilities and factor graphs decompose the joint Density into a product of factors. Decomposing the density is beneficial for several reasons:

- **Pragmatic:** the modeler can often express the joint density as a composition of common Densities which are built into the modeling language and which are easy for another human to interpret
- **Learnability:** decomposing the joint density often reduces the number of degrees of freedom. For example, expressing the joint density over two boolean variables x and y as a single conditional probability table would require 3 parameters, e.g. $p(x \wedge y)$, $p(x \wedge \neg y)$, and $p(\neg x \wedge y)$, with $p(\neg x \wedge \neg y) = 1 - p(x \wedge y) - p(x \wedge \neg y) - p(\neg x \wedge y)$; in contrast, if the joint probability can be

decomposed into two Densities, each of which depends on only one of the variables (e.g., if x and y are independent), then only two parameters are needed, e.g. $p(x)$ and $p(y)$, with $p(\neg x) = 1 - p(x)$ and $p(\neg y) = 1 - p(y)$.

- **Efficiency:** decomposing the joint density often allows the joint density to be computed more quickly, by avoiding the re-evaluation of Densities for which the dependent States have not changed, or by exploiting parallelism (computing multiple Densities simultaneously on different processing units)
- **Algorithmic:** the decomposition of the joint density determines which variables in the state space are conditionally independent. Some algorithms can exploit this independence in order to be more efficient in time or storage requirements, or operate on multiple segments of the state space in parallel. For instance, in a factor graph, two variables a and b are conditionally independent given a set of variables $C \subseteq Vars - \{a\} - \{b\}$, if every path from a to b contains at least one variable from C . It is therefore possible to use Gibbs sampling, for example, to resample a and b simultaneously and independently if C is (temporarily) held constant.

Just as the BLAISE State representation can be viewed as an extension of the Bayes net/factor graph representation of variables, the BLAISE Density representation is an extension of the factor nodes in a factor graph. Like factor nodes, BLAISE Densities are graph nodes that have edges connecting them to each of the States on which the value of the Density depends. For example, see figure 3-7.

Unlike factor graph nodes, however, BLAISE Densities are structured into a tree. In addition to Density→State edges, a Density might also have edges connecting it to other Densities, the value of which it might depend upon. These Density→Density edges form a rooted tree where the root node represents the joint probability density function for the entire state space. Leaf nodes in the Density tree typically represent common (primitive) probability densities, such as the Beta and Binomial Densities in figure 3-7. Internal nodes in the Density tree represent functional composition of Densities. Whereas the only functional composition rule permitted (implicitly) in factor graphs is multiplication (i.e. the total joint density is the product of the

factors), the BLAISE modeling language allows great freedom in the functional form of a Density. Specifically, if a Density d has links to States S_1, \dots, S_n and links to child Densities C_1, \dots, C_m , the value of the Density may be computed by any function f of the form

$$\text{density}(d) = f(\text{density}(C_1), \dots, \text{density}(C_m); S_1, \dots, S_n).$$

with the explicit interpretation that depending on state S_i implies that the density may inspect S_i and any of its descendants.

Most internal (composite) Density nodes will be simple Multiplicative Densities, which will have no State links and which have the functional form

$$f(\text{density}(C_1), \dots, \text{density}(C_m); S_1, \dots, S_n) = \prod_{i=1}^m \text{density}(C_i)$$

Leaf (atomic) Density nodes have no Density children, so their functional form is

$$f(\text{density}(C_1), \dots, \text{density}(C_m); S_1, \dots, S_n) = p(S_1, \dots, S_n)$$

where p is a local contribution to the joint density.

However, there exist useful composite Densities that are not multiplicative, or even linear. For example, in parallel tempered inference or simulated annealing, it is necessary to adjust the “temperature” of a distribution by exponentiating the density; this can be accomplished with a Density that has one dependent State S_τ (the temperature) and one child Density C_1 , and is evaluated using

$$f(\text{density}(C_1); S_\tau) = \text{density}(C_1)^{1/S_\tau}.$$

BLAISE’s support for non-linear Density composition, and specifically the tempering Density shown above, will be highlighted in sections 4.2.1–4.2.3.

One side effect of allowing such non-linear density functions is that if D_{root} is the root density and $D_{descendant}$ is some density with D_{root} as an ancestor, then changes to the value of $\text{density}(D_{descendant})$ can have an arbitrary effect on the value

of $density(D_{root})$. In order to determine this effect, it will be necessary to consider the effect of each Density on the path from $D_{descendant}$ to D_{root} . It is one of the hypotheses of this thesis that all density calculations required for standard inference can be expressed as evaluations of D_{root} , even if it is known that only a smaller portion of the Density tree has links to the State currently being updated by the inference algorithm. This hypothesis will be supported as the BLAISE treatments of the Metropolis-Hastings method, the Gibbs sampling method, and particle filters are presented.

In section 3.2, it was noted that many models use repeated structures in the State space. It is usually the case that these repeated State structures give rise to functionally identical terms in the joint density. For example, reconsider a simple model for fitting a beta-binomial model. Figure 3-7 showed this model for a single datapoint x , and figure 3-5 showed the state space for this model for multiple datapoints. To update the Density structure from the single datapoint model to accommodate the multiple datapoint statespace, the first step is to create a binomial Density for each datapoint, connected to the datapoint and to the shared parameter p . To compose these Densities, BLAISE supports Multiplicative Collection Densities, which are the Density analog to Collection States. A Multiplicative Collection Density composes any number of child Densities by computing the product of their values (see figure 3-8).

3.4 BLAISE Kernels

A State–Density graph describes a complete probabilistic model in terms of a joint density over a state space. While existing probabilistic modeling languages typically stop here, BLAISE goes one step farther by also graphically representing the inference procedure that will be executed on the model. BLAISE focuses on those inference techniques that can be described as history-free stochastic walks through a State space, guided by a Density².

²Restricting inference to methods that can be implemented as history-free stochastic walks is actually not as restrictive as it may first appear, given that deterministic walks are a subset of

The central abstraction for inference procedures in BLAISE is the Transition Kernel, typically abbreviated to just *Kernel*³. Mathematically, a transition kernel is an expression of the probability that the stochastic walk will make a particular step in the state space, given the state the walk is currently at. That is, if the walk is currently at State S_t , a transition kernel will specify, for any State S_* in the state space, the probability that the next state $S_{t+1} = S_*$. This probability could be written $p(S_{t+1} = S_*|S_t)$; however, this thesis will use the alternate notation $K(S_t \rightarrow S_*)$ to emphasize the directionality of the transition being evaluated⁴.

In BLAISE graphical models, a Kernel is represented as a small triangle. The symbol was chosen to bring to mind the capitalized Greek letter delta (Δ), because delta is the traditional symbol for change and Kernels update the walk to the next State. Each Kernel operates on some subgraph of the State hierarchy. Most Kernels operate on only a single State and may only inspect or modify that State and its descendents. The State that the Kernel operates on is indicated graphically using a directed edge from the Kernel to the State. For example, a Kernel that will resample the value of a continuous State variable (for example, p in the beta-binomial model in section 3.3) must have a directed edge to that State or one of its ancestors. More complex Kernels may use multiple Kernel→State edges, indicating that the Kernel operates on a sub-forest of the State graph (rather than a simple sub-tree in the single-edge case). For example, if a mixture model’s components were represented as Collection States as in figure 3-6, then a datapoint reassignment Kernel would change the assignment of a datapoint by removing the datapoint from one compo-

stochastic walks, and that the state space may be augmented with any information that would normally be considered the history of the kernel. See chapter 4 for examples of state space augmentation; for example, the simulated annealing transform in section 4.2.2 augments the state space with a temperature variable, using a deterministic Kernel to decrease the temperature slowly over the course of inference.

³It should be emphasized that, despite their name, Transition Kernels are unrelated to the kernels used in “kernel methods” such as support vector machines, or to kernels of homomorphisms in algebra, etc.

⁴In the author’s experience, even those who are experienced practitioners of MCMC-based inference make frequent mistakes when using the traditional notation $p(S_*|S_t)$, particularly when working with quantities such as the Metropolis-Hastings acceptance ratio, which includes both the terms $p(S_*|S_t)$ and $p(S_t|S_*)$. The alternate notation, e.g. $K(S_t \rightarrow S_*)$ and $K(S_* \rightarrow S_t)$ produces significantly less confusion.

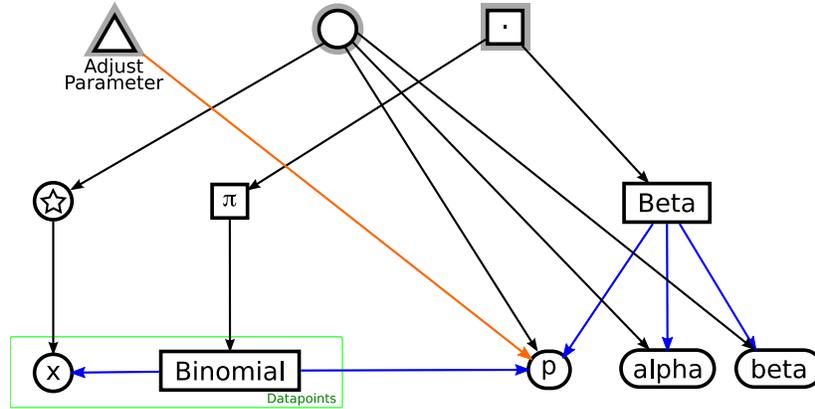
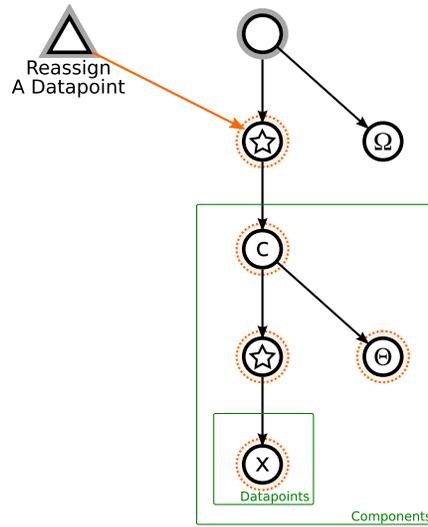


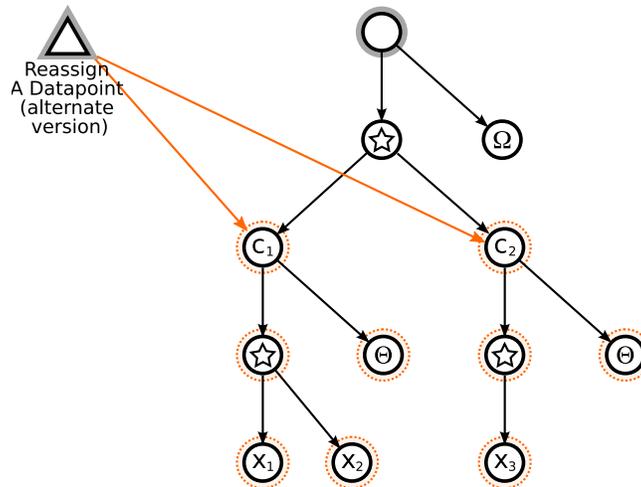
Figure 3-9: The beta-binomial model from figure 3-8, with a Kernel to perform inference on the parameter p . A Kernel is represented as a triangle with links to the States it may inspect and modify. The gray annulus around the parameter adjustment kernel in this example highlights it as the root Kernel.

ment’s Collection State and adding it to another component’s Collection State. Such a Kernel could be implemented with a single edge to a common ancestor of the two components, as in figure 3-10a. Alternately, the Kernel could be implemented with two edges: one to the source component and one to the target component, as in figure 3-10b. The latter implementation allows the Kernel to be reused more flexibly by separating the datapoint reassignment logic from the component selection logic.

Kernels also have limited access to the Density graph: Kernels may evaluate the root node of the Density tree, but may not inspect the Density tree in any other way. Specifically, Kernels may not inspect the structure of the Density tree, nor may they modify the Density tree, nor may they evaluate any node but the root node of the Density tree. These restrictions are motivated by two points: first, as stated in section 3.3, it is part of the hypothesis of this thesis that all density calculations required for standard inference can be couched as evaluations of the root Density node. Second, it is a central design goal for BLAISE to support composition of models, including inference algorithms on those models, and further including that it should be possible to mix-and-match fragments of models with minimal effort. If the Kernel were permitted to inspect the Density tree, it would be much more difficult to perform these types of composition. For further justification of these design choices, see section 3.9 for a comparison to the Model-View-Controller design pattern.



(a) Single Link Kernel



(b) Dual Link Kernel

Figure 3-10: As described in figure 3-6, datapoint reassignment Kernels for BLAISE mixture models move datapoints from one Collection State to another. (a) shows a datapoint reassignment Kernel that uses a single edge to find the Collection of components. (b) shows an alternate form for a datapoint reassignment Kernel that uses two edges to locate the two components that will be the source and destination for the moving datapoint (the expanded form of the BLAISE model is used here to show that the Kernel is linking to two distinct components). In these two diagrams, each State that may be inspected or modified by the Kernel is highlighted with a dotted orange outline.

Every BLAISE Kernel provides a SAMPLE-NEXT-STATE operation; this operation considers the current state S_t and samples a next state S_* for the stochastic walk from the transition distribution encoded in the kernel, i.e. $S_* \sim K(S_t \rightarrow S_*)$. Standard Markov chain Monte Carlo inference in BLAISE, then, is a matter of:

1. initializing the State structure so that it is in the domain and matches the observed evidence
2. repeatedly calling SAMPLE-NEXT-STATE on an appropriate Kernel (i.e. a Kernel with the correct invariant distribution)
3. recording the states visited by the stochastic walk as samples from the target distribution.

The observed variables should be held constant either by attaching Kernels only to the unobserved variables or by attaching Dirac delta Densities to the observed variables (such that any value but the observed value causes the Density to evaluate to 0).

BLAISE Kernels may also support two optional operations: SAMPLE-NEXT-MOVE and ENUMERATE-POSSIBLE-MOVES. SAMPLE-NEXT-MOVE is much like SAMPLE-NEXT-STATE, except that instead of producing just a sample S_* , SAMPLE-NEXT-MOVE produces a Move object: $S_t \xrightarrow{K} S_*$. A Move object carries several pieces of information. The next state is still available:

$$\text{MOVE-TARGET}(S_t \xrightarrow{K} S_*) \triangleq S_*.$$

Move objects also carry additional information, such as the probability that the Kernel's SAMPLE-NEXT-MOVE will produce this move:

$$\text{MOVE-FORWARD-TRANSITION-DENSITY}(S_t \xrightarrow{K} S_*) \triangleq K(S_t \rightarrow S_*)$$

and the probability that the Kernel's SAMPLE-NEXT-MOVE would produce the inverse move from the target state:

$$\text{MOVE-REVERSE-TRANSITION-DENSITY}(S_t \xrightarrow{K} S_*) \triangleq K(S_* \rightarrow S_t).$$

Operations for a Kernel K

SAMPLE-NEXT-STATE(S_t)	Sample $S_* \sim K(S_t \rightarrow S_*)$, returning S_*
SAMPLE-NEXT-MOVE(S_t)	Sample $S_* \sim K(S_t \rightarrow S_*)$, returning a Move $S_t \xrightarrow{K} S_*$ (optional)
ENUMERATE-POSSIBLE-MOVES(S_t)	Return a collection of all Moves $S_t \xrightarrow{K} S_*$ that could be generated by SAMPLE-NEXT-MOVE (optional)

Figure 3-11: The operations supported by a Kernel K . Some operations are optional, though some Kernels may require that their subkernels support a particular optional operation. For example, a Metropolis-Hastings Kernel requires that its proposal Kernel supports SAMPLE-NEXT-MOVE. Other Kernels may only support optional operations if their children support the appropriate operations as well. For example, a concrete cycle Kernel supports ENUMERATE-POSSIBLE-MOVES only if all its subkernels support ENUMERATE-POSSIBLE-MOVES.

Finally, in order to support transdimensional MCMC, Moves also carry information about the Jacobian of the Move under the Kernel, accessible via MOVE-JACOBIAN⁵.

SAMPLE-NEXT-MOVE enables the fully-generic implementation of algorithms such as Metropolis-Hastings, as will be discussed in section 3.4.2, and particle filtering, as will be discussed in section 4.3. Note that any Kernel implementing SAMPLE-NEXT-MOVE can implement SAMPLE-NEXT-STATE simply as

$$\text{SAMPLE-NEXT-STATE}() = \text{MOVE-TARGET}(\text{SAMPLE-NEXT-MOVE}).$$

The other optional operation of a Kernel is ENUMERATE-POSSIBLE-MOVES, which produces the set of all possible Move objects that that could be returned by a call to SAMPLE-NEXT-MOVE. Note that implementing ENUMERATE-POSSIBLE-MOVES may be impossible; for example, if the the Kernel operates on continuous variables, it probably can produce an infinite number of distinct moves, and thus can't implement ENUMERATE-POSSIBLE-MOVES. ENUMERATE-POSSIBLE-MOVES

⁵Although all Moves support the MOVE-JACOBIAN operation, in many cases it will simply evaluate to 1. In particular, if the methods defined in the Kernel do not require the use of reversible-jump MCMC (for example, if they do not change the dimension or parameterization of the State space), then the Kernel's Moves' Jacobian will be 1.

Operations for a Move $S_t \xrightarrow{K} S_*$

MOVE-TARGET($S_t \xrightarrow{K} S_*$)	return S_*
MOVE-FORWARD-TRANSITION-DENSITY($S_t \xrightarrow{K} S_*$)	return $K(S_t \rightarrow S_*)$
MOVE-REVERSE-TRANSITION-DENSITY($S_t \xrightarrow{K} S_*$)	return $K(S_* \rightarrow S_t)$
MOVE-JACOBIAN($S_t \xrightarrow{K} S_*$)	return this Move's Jacobian ⁶

Figure 3-12: The operations supported by a Move $S_t \xrightarrow{K} S_*$. Moves are generated from Kernels via the SAMPLE-NEXT-MOVE and ENUMERATE-POSSIBLE-MOVES operations to carry information about the sampling process executed by the Kernel.

enables the fully-generic implementation of algorithms such as Gibbs sampling for enumerable variables, as discussed in section 3.4.2. Any Kernel that implements ENUMERATE-POSSIBLE-MOVES can implement SAMPLE-NEXT-MOVE simply by sampling from the the set of Moves produced by ENUMERATE-POSSIBLE-MOVES, with each Move $S_t \xrightarrow{K} S_*$ sampled with MOVE-FORWARD-TRANSITION-DENSITY($S_t \xrightarrow{K} S_*$).

Like States and Densities, Kernels are also composed into trees. In the case of Kernels, the tree structure represents algorithmic composition: a Kernel may call operations on any of its child Kernels any number of times as part of its operation⁷. Composition Kernels may also be viewed as analogous to the control-flow operations in other programming languages (e.g. `for`, `case`, `if-then-else`, etc.), including stochastic generalizations of these constructs.

3.4.1 Hybrid Kernels

Hybrid Kernels are the most common composite Kernels because they are stationary distribution-preserving; that is, if all of a hybrid Kernel's child Kernels share a stationary distribution on the State space, then the hybrid Kernel is guaranteed to share that stationary distribution as well. As described in chapter 2, the two standard hybrid Kernels are the cycle and mixture hybrids. A concrete⁸cycle Kernel can have an arbitrary number of child Kernels and implements SAMPLE-NEXT-STATE by calling

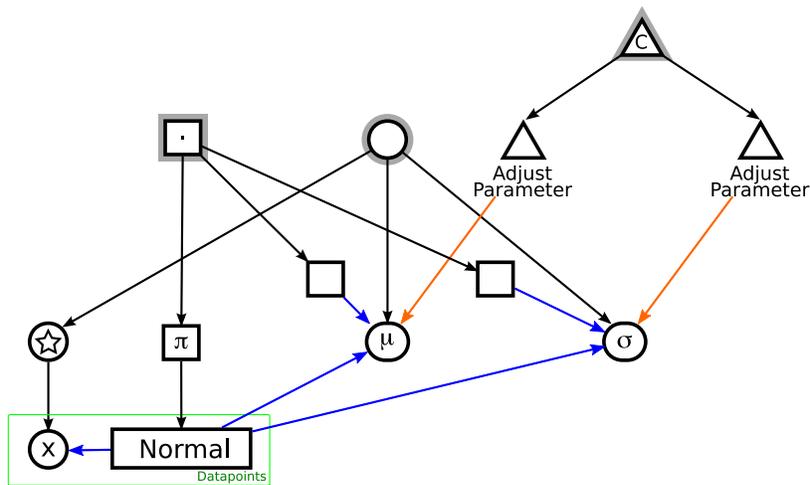
⁷Only it's immediate children – a Kernel may *not* invoke operations on other descendants.

SAMPLE-NEXT-STATE on each of its child Kernels one after another. If the child Kernels are unrelated, the resulting sequential control flow will be similar to a series of statements in an imperative language like Java or C, or like the body of a `begin` statement in Scheme. If, instead, each child Kernel performs the same operation but targets a different State, the resulting control flow is akin to a `for` statement (though see section 3.6 for an even closer relative to the `for` statement). Figure 3-13a shows an example of concrete cycle Kernels in BLAISE SDK diagrams.

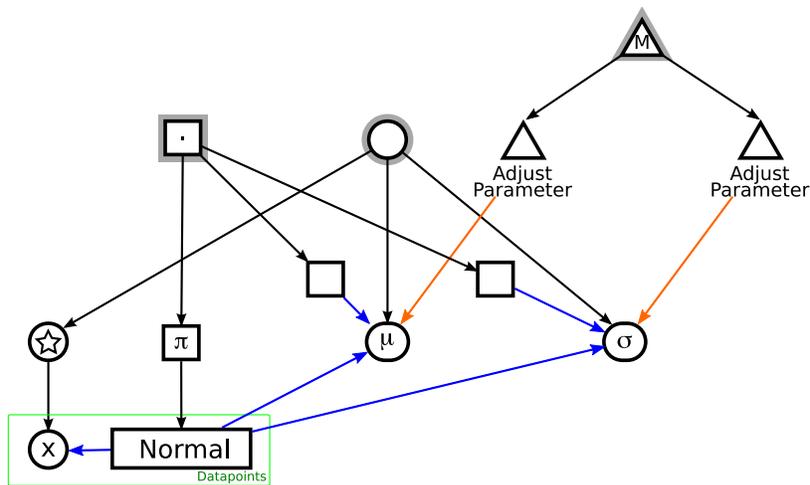
A concrete mixture kernel has an arbitrary number of child Kernels and associates a weight with each child Kernel; when the hybrid Kernel’s SAMPLE-NEXT-STATE operation is called, the Kernel first selects one of its child Kernels, sampled proportional to their weights, then delegates to the selected child’s SAMPLE-NEXT-STATE method. The resulting control flow is analogous to a `case` statement, where the expression being switched upon is a random number drawn according to the child Kernels’ weights. Figure 3-13b shows an example of concrete mixture Kernels in BLAISE SDK diagrams.

BLAISE also introduces a novel class of hybrid Kernels: conditional hybrid Kernels. A (binary) conditional hybrid Kernel has two child Kernels; a `TRUE`-Kernel and a `FALSE`-Kernel. It also has a deterministic binary predicate that is defined over the Kernel’s operating State space. A conditional hybrid Kernel interprets calls to SAMPLE-NEXT-STATE by evaluating the predicate, then delegating to the child Kernel associated with the predicate’s result (i.e. if the predicate evaluates to `TRUE`, then the conditional hybrid Kernel delegates to its `TRUE`-Kernel’s SAMPLE-NEXT-STATE operation.) Conditional hybrid Kernels are not restricted to binary predicates; the predicate may be replaced with any deterministic function of the State, so long as the conditional hybrid Kernel can map any value of the function to exactly one child Kernel. Appendix A proves that, if all the children of a conditional hybrid Kernel share a stationary distribution, and if no child can change the value of the conditional hybrid Kernel’s predicate/expression, then the resulting conditional hybrid Kernel is guaranteed to have the same stationary distribution as its children. The control flow

⁸Virtual hybrid Kernels, including virtual cycle Kernels and virtual mixture Kernels, will also be introduced in section 3.6.



(a) Cycle Hybrid



(b) Mixture Hybrid

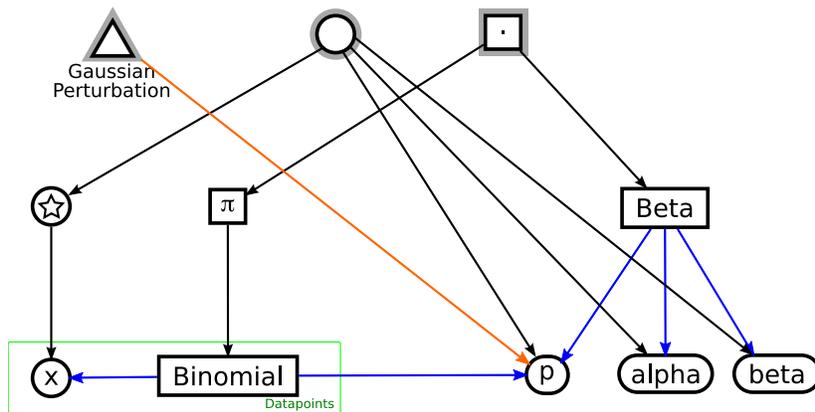
Figure 3-13: A Kernel for performing parameter inference in a single parameter beta-binomial model was shown in Figure 3-9. What if the model were instead a Gaussian (normal) model, with two parameters (μ and σ) to be inferred? Figures (a) and (b) show such a model (hyperparameters on μ and σ are omitted for legibility, but the Densities representing the priors on μ and σ are depicted). Each model has one parameter inference Kernel for each parameter. In (a), a concrete cycle Kernel, labeled “C”, composes the two parameter inference Kernels into a single Kernel structure; on each step of the walk, both parameters will be adjusted. Other hybrid Kernels are also possible. For example, in (b) the concrete cycle Kernel is replaced by a concrete mixture Kernel, labeled “M”; on each step in the walk, the mixture Kernel randomly chooses to update either μ or σ .

resulting from a conditional hybrid Kernel is much like an `if` statement or a `case` statement.

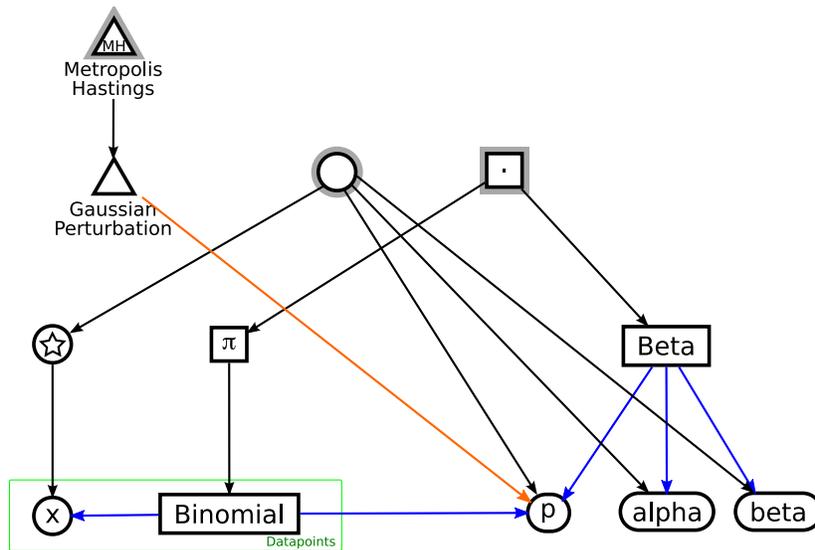
3.4.2 Metropolis-Hastings and Gibbs Sampling

The Metropolis-Hastings (MH) method, as described in chapter 2, is a popular method for producing kernels with the desired stationary distribution. In BLAISE, the MH method is implemented generically using Kernel composition; the user supplies a proposal Kernel defined on the State space, then uses a BLAISE-standard generic MH Kernel that has the proposal Kernel as a child. The MH Kernel uses `SAMPLE-NEXT-MOVE` from its proposal Kernel to make a proposal. The acceptance ratio can then be evaluated using the operations defined by the Move in concert with evaluations of the Density tree. Finally, the MH Kernel either accepts or rejects the proposal. (As a side note, the BLAISE MH Kernel is actually a reversible-jump MH Kernel, though in the common case where the proposal Kernel returns Moves whose Jacobians are 1, the reversible-jump method recovers the standard MH method. See figure 3-15 for pseudo-code for the MH Kernel).

Gibbs sampling, as described in chapter 2, is a common subcase of the MH method that usually produces efficient MH kernels. BLAISE supports Gibbs sampling as a generic method when the values of the State to be resampled can be efficiently enumerated as a discrete set. Much like the BLAISE implementation of MH, the user supplies an “enumeration” Kernel (a Kernel defined on the appropriate State and supporting the `ENUMERATE-POSSIBLE-MOVES` operation, also called a proposal Kernel by analogy to MH), then specifies a BLAISE-standard generic Gibbs Kernel that has the enumeration Kernel as a child. The Gibbs Kernel uses `ENUMERATE-POSSIBLE-MOVES` from the enumeration Kernel to produce a set of Moves to all possible values of the State being resampled; the Gibbs Kernel then uses evaluations of the Density tree to weight each of these Moves (ignoring the transition probabilities assigned to the Moves by the enumeration Kernel). Finally, the Gibbs Kernel samples one of the Moves proportional to the weights. (See figure 3-17 for pseudo-code for the Gibbs Kernel).



(a) This model exhibits Brownian motion



(b) This model samples from interest distribution

Figure 3-14: Metropolis-Hastings (MH) Kernels in BLAISE are generic adapters. In figure (a), a Gaussian Perturbation Kernel is being used to adjust a parameter by sampling from a Gaussian distribution centered on the current value of that parameter: $K_{gp}(s_t \rightarrow s_*) = Normal(p = p_*; \mu = p_t, \sigma = \sigma_0)$. This configuration produces Brownian motion on the parameter p , resulting in diffusion rather than convergence to the interest distribution (i.e. the joint density induced by the Density structure). In figure (b), a generic MH Kernel has been added to the model. This Kernel uses the Gaussian Perturbation Kernel to generate proposal Moves. These moves are then accepted or rejected using the standard Metropolis-Hastings acceptance ratio calculations, resulting in an MCMC chain that converges to the interest distribution $p(p|\alpha, \beta, \vec{x})$.

```

METROPOLIS-HASTINGS.SAMPLE-NEXT-STATE(State, Root-Density)
1  move ← Proposal-Kernel.SAMPLE-NEXT-MOVE(State, Root-Density)
2  densitypre ← Root-Density(State)
3  densitypost ← Root-Density(MOVE-TARGET(move))
4  proposalforward ← MOVE-FORWARD-TRANSITION-DENSITY(move)
5  proposalreverse ← MOVE-REVERSE-TRANSITION-DENSITY(move)
6  paccept ← min  $\left(1, \frac{\textit{density}_{\textit{post}}}{\textit{density}_{\textit{pre}}} \cdot \frac{\textit{proposal}_{\textit{reverse}}}{\textit{proposal}_{\textit{forward}}} \cdot \textit{MOVE-JACOBIAN}(\textit{move})\right)$ 
7  accepted ∼ BERNOULLI(paccept)
8  if accepted
9      then return MOVE-TARGET(move)
10     else return State

```

Figure 3-15: Pseudocode for a Metropolis-Hastings Kernel’s SAMPLE-NEXT-STATE.

The MH and Gibbs Kernels are important examples of the BLAISE framework for several reasons. First, they illustrate Kernel composition, using composition strategies other than the standard hybrid Kernels. They also illustrate an “adapter” pattern that is common in BLAISE, as will be seen in the discussion of State–Density–Kernel (SDK) transformations in chapter 4. In fact, Metropolis-Hastings and (enumerative) Gibbs sampling can be considered two of the most basic transformations supported in BLAISE. The “MH transformation” takes as input an SDK graph, where the Kernel supports the operation SAMPLE-NEXT-MOVE but does not have as a stationary distribution the Density’s distribution over the State space. By transforming the SDK graph – that is, by wrapping the existing Kernel in an MH Kernel – the MH transformation produces an SDK graph in which the Kernel does have the correct stationary distribution. Likewise, the “Gibbs transformation” takes as input an SDK graph, where the the Kernel supports the operation ENUMERATE-POSSIBLE-MOVES but does not have as a stationary distribution the Density’s distribution over the State space. By transforming the SDK graph – that is, by wrapping the existing Kernel in a Gibbs Kernel – the Gibbs transformation produces an SDK graph in which the Kernel does have the correct stationary distribution and is efficient.

Because MH and Gibbs Kernels use regular BLAISE Kernels to describe their proposals, BLAISE’s composition tools can be used to form interesting proposals. For

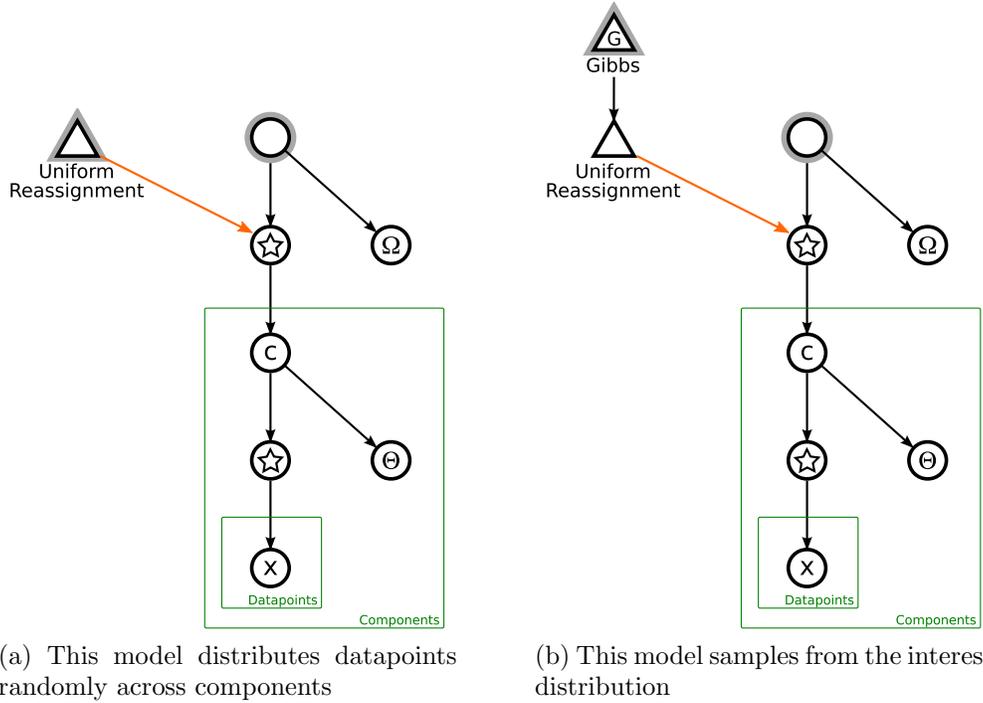


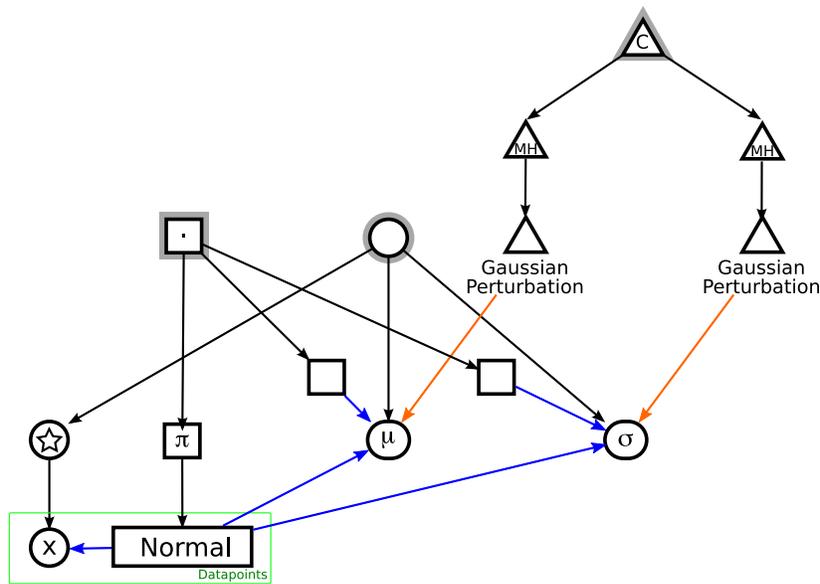
Figure 3-16: Gibbs Kernels in BLAISE are generic adapters. In figure (a), the Uniform Reassignment Kernel’s SAMPLE-NEXT-STATE selects a datapoint uniformly from the set of all datapoints, then selects a mixture component uniformly from the set of components, and finally moves the datapoint out of its current component and into the selected component. This configuration will produce a random shuffling of the datapoints, distributing them uniformly across the components without regard the value of the datapoints, rather than converging to the interest distribution. In figure (b), a generic enumerative Gibbs Kernel has been added to the model. This Kernel uses the Uniform Reassignment Kernel’s ENUMERATE-POSSIBLE-MOVES to generate candidate Moves. The Gibbs Kernel then selects from these Moves, weighting each move using the joint density of its target state, resulting in an MCMC chain that converges to the interest distribution.

```

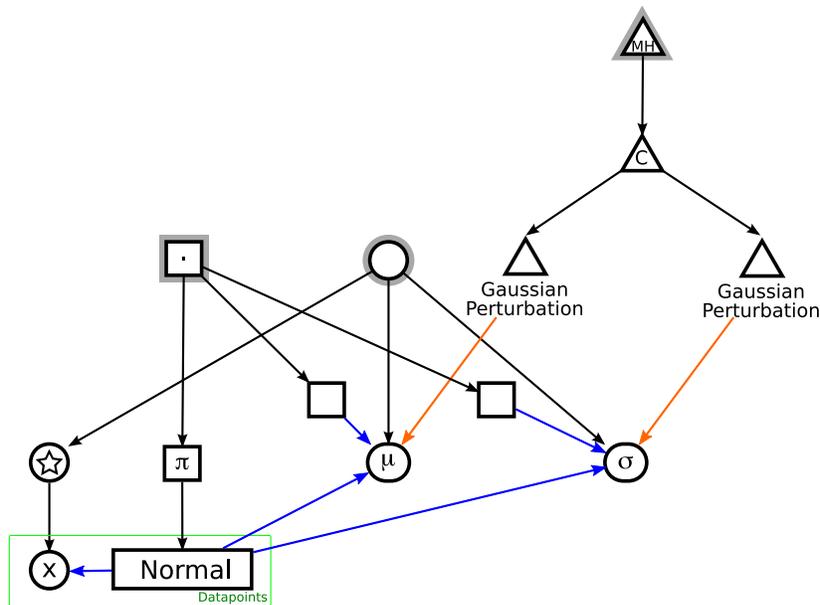
GIBBS.SAMPLE-NEXT-STATE(State, Root-Density)
1  moves[] ← Proposal-Kernel.ENUMERATE-POSSIBLE-MOVES(State, Root-Density)
2  for each move ∈ moves[]
3      do density[move] ← Root-Density(MOVE-TARGET(move))
4  densitysum ← ∑move density[move]
5  move ~ moves[] weighted by  $\frac{\text{density}[\text{move}]}{\text{density}_{\text{sum}}}$ 
6  return MOVE-TARGET(move)

```

Figure 3-17: Pseudocode for a Gibbs sampling Kernel’s SAMPLE-NEXT-STATE.



(a) Unblocked



(b) Blocked

Figure 3-18: Kernel composition can be used to create interesting proposal Kernels. Figure (a) shows a standard cycle Kernel composition of two unblocked Metropolis-Hastings samplers for the parameters of a Normal model. In figure (b), the cycle Kernel has been “pushed through” the Metropolis-Hastings Kernel, yielding a blocked sampler that will resample both μ and σ together. Blocked Metropolis-Hastings samplers allow larger steps to be taken in the state space and may produce a Markov chain that mixes better (e.g. by making it more likely to move between two modes); however, these larger moves can also negatively impact mixing (e.g. by making it more likely that moves will be proposed into low probability regions of the state space).

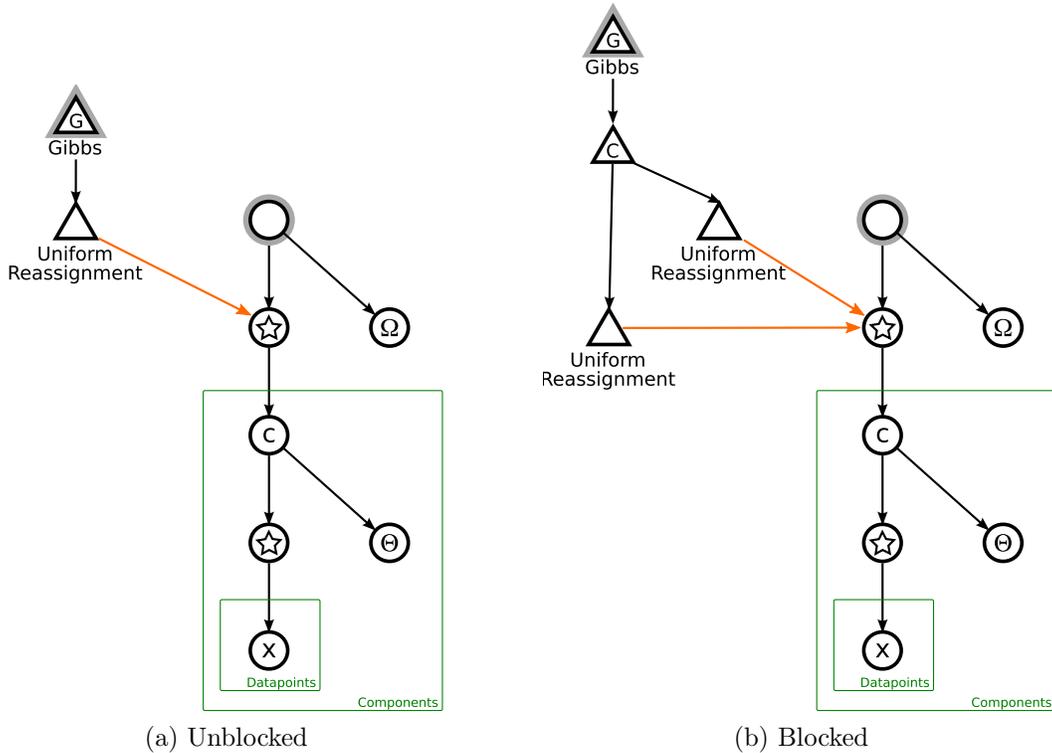


Figure 3-19: Kernel composition can be used to create interesting proposal Kernels. Figure (a) shows a standard Gibbs sampler for reassigning datapoints in a mixture model. In figure (b), a cycle Kernel has been introduced into the Gibbs Kernels' proposal, yielding a blocked sampler that will resample assignments for two datapoints together. Blocked Gibbs samplers allow larger steps to be taken in the state space and may produce a Markov chain that mixes better; however, blocking also exponentially increases the number possible moves that must be evaluated.

example, using a cycle hybrid as the proposal for an MH or Gibbs Kernel will result in blocked MH or Gibbs, respectively, where several variables are resampled as a group or “block.” These patterns are demonstrated in figures 3-18 and 3-19.

3.5 Densities for state spaces of unknown dimensionality or variable structure

The BLAISE tools described so far would be sufficient if the topology of BLAISE State spaces were static; however, to fully support mutable State spaces, a few more tools are needed. Specifically, BLAISE should be able to model how the Densities and Kernels accommodate State spaces that change.

Consider, for example, Collection States that may have children added and removed at inference time, such as those representing the components in a beta-binomial mixture model. As data points are added to a mixture component, the structure of the density of that component should be updated to reflect the addition. As per the previous discussion of beta-binomial models, the Density of a component could be represented as a Multiplicative Collection Density, with one child Binomial Density per datapoint. Keeping the Density up-to-date then requires a new Binomial Density to be created, to be connected to the data point and the component’s parameter p , and to be added as a child of the component’s Collection Density. An important question remains: where does the responsibility for this update lie? One option would be for the Kernel that modifies the State space topology to make sympathetic modifications to the Density topology; however, section 3.4 already concluded that this strategy of directly coupling Kernels and Densities is detrimental to compositionality, preferring instead to let the State space mediate between them. This leaves two clear options: either the State space can respond to changes by mutating the Density hierarchy, or else Densities can watch the State space for changes and update their own structures accordingly. The difference is in where the information and responsibility for updates resides: in the former case, the information resides in the State space, whereas in the

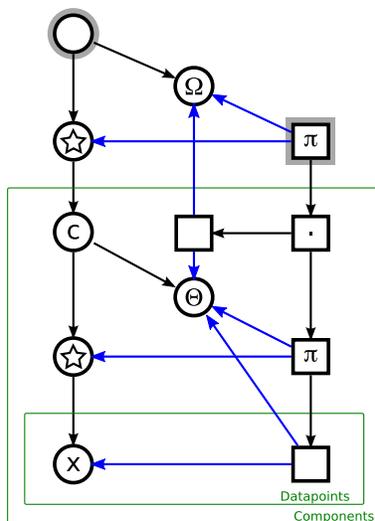


Figure 3-20: Associated Collection Densities allow the Density structure to maintain consistency with the State structure, even as States are added to and removed from Collection States. An Associated Collection Density has a Density→State edge to a Collection State, ensuring a one-to-one correspondence between children of the Collection State and children of the Associated Collection Density. The Associated Collection will construct and destroy child Densities as needed to satisfy this contract. This figure shows the State and Density structure for a mixture model, using two Associated Collection Densities: one to make sure that every datapoint x has a likelihood Density connecting it to the component parameter Θ , and one to ensure that each component has the appropriate Density structure (i.e. the aforementioned datapoint Associated Collection Density, a prior Density connecting Θ to Ω , and a Multiplicative Density to compose the two). This Density structure will stay consistent as Kernels move datapoints from one component to another, and even if components are created or destroyed.

latter case, the information resides in the Density hierarchy. Given that the State space, as described earlier in this chapter, has no dependencies whatsoever on Densities and their structure, and given that the Density structure is already intimately connected with the structure of the State space via the Density→State links, the choice seems clear: in BLAISE, Densities may watch the State space and mutate their own structure in response to changes in the State space.

Let us now use this premise to complete the Density structure for a mixture model. We wish to ensure that a component’s Multiplicative Collection Density has a Density associated with each datapoint in the component’s Collection State; therefore, the Multiplicative Collection Density should watch the Collection State and add (or

remove) a child Density each time the Collection State has a child datapoint added (or removed). To note this graphically, we reuse the notation for Density→State dependencies and add an edge from the Multiplicative Collection Density to the Collection State, as in figure 3-20. This pattern is so common in BLAISE models that the Multiplicative Collection Density implementing it is given a special name: an Associated Collection Density (because it associates a child with every child of a Collection State). The Density story is now almost complete, with just one thing remaining: in order for the Associated Collection Density to create the datapoint Density children, it will need to be able to locate the component parameter State. Therefore, the Associated Collection Density has a link to the component parameter Θ – not because the Collection Density’s value directly depends on Θ , nor because it is going to mutate its structure in response to Θ , but because it will need to find Θ to construct the datapoint Densities.

3.6 Kernels for state spaces of unknown dimensionality or variable structure

As State spaces vary in dimension, it is also important to ensure that Kernels are dispatched appropriately to explore the entire State space. For example, in the mixture model example, it is important to make sure that Kernels for component parameter inference are applied to each of the components, no matter how many components exist. If the number of components is known a priori, the designer can simply use a concrete hybrid kernel (either a mixture or a cycle). However, what if determining the number of components is one of the inference goals?

BLAISE introduces a novel kind of Kernel, called a virtual hybrid Kernel, to manage Kernel dispatch over Collection States. A virtual hybrid Kernel can be thought of as a concrete hybrid Kernel that has, as children, a copy of a subkernel for each element of the Collection State. For example, a virtual cycle Kernel for the components of a mixture model would act as if it had one copy of the component-parameter-

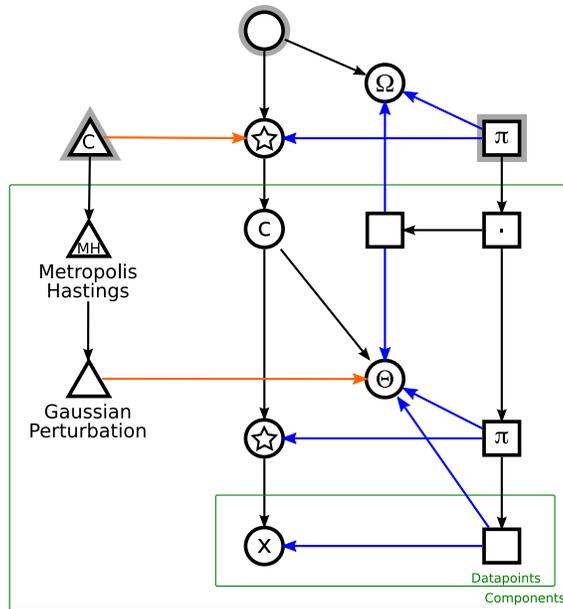


Figure 3-21: As the number of components in a mixture model changes, the number of component parameters requiring inference also changes. BLAISE uses virtual hybrid Kernels, like the virtual cycle Kernel labeled “C” in this diagram, to represent a hybrid Kernels that operate over an unknown number of States. This virtual cycle Kernel will apply the Metropolis-Hastings-adapted Gaussian Perturbation Kernel to the Θ parameter of each component in turn.

adjustment-kernel for each component in the mixture. When a component is added, the virtual cycle acts as if it has had added a new component-parameter-adjustment-kernel with an edge to the new component.

Virtual hybrid Kernels are called “virtual” because they only actually need one copy of the child Kernel; rather than making many copies of the child Kernel, the virtual hybrid just calls the same child Kernel multiple times, directing it at a different state each time. Virtual hybrid Kernels are possible because Kernels are history-free, that is, stateless⁹.

Like concrete hybrid Kernels, virtual hybrid Kernels have the property that the hybrid Kernel shares the same stationary distribution as its child Kernel, so long as it can be guaranteed that the child Kernel is unable to change the number of

⁹In comparison, some machines for executing BLAISE SDK automata might treat Densities as mildly stateful once they are attached to the State hierarchy, in that different copies of the same Density, if connected to different States, will have different values. For example, chapter 5 will describe how Densities in the BLAISE Virtual Machine maintain some state in the form of a cache that significantly improves the performance of Density evaluation.

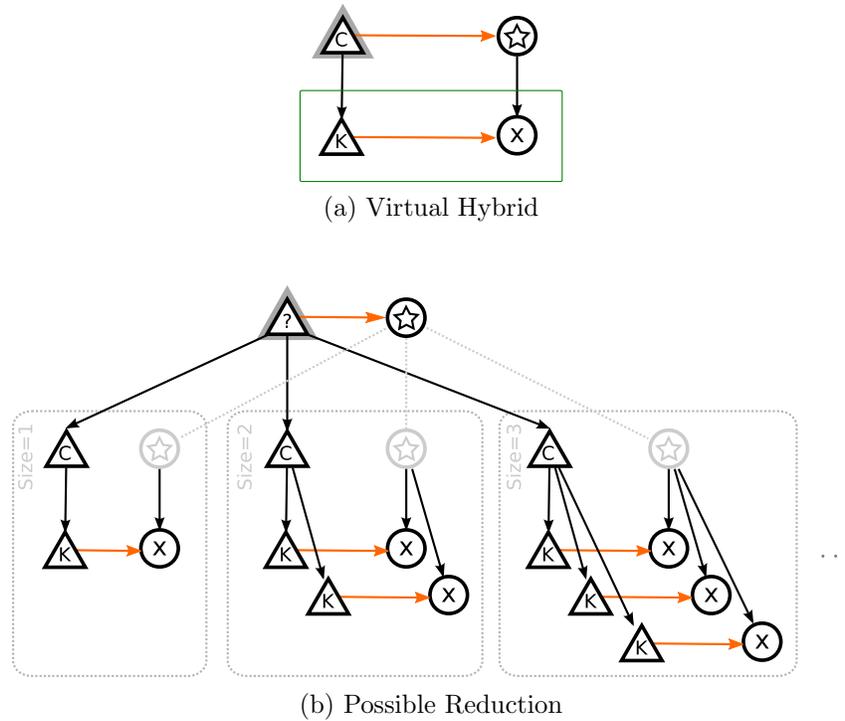


Figure 3-22: Virtual Hybrid Kernels can be analyzed as a Conditional Hybrid of Concrete Cycles, as described in section 3.6. Figure (a) shows a simple Virtual Hybrid Kernel configuration. Figure (b) shows a hypothetical reduction of the virtual hybrid configuration, using a conditional hybrid Kernel (labeled “?”) that partitions that state space based on the number of children in the Collection State. The dotted gray boxes show example states that would fall into the partition subspaces for Collections States of size 1, 2, and 3, with the ellipsis indicating the omission of larger subspaces from the diagram. The gray Collection State depicted in each subspace would not actually exist in the real model; it is just a placeholder for the actual Collection State that would exist in all the subspaces and which is depicted beside the Conditional Hybrid Kernel.

children in the Collection State that the virtual hybrid Kernel is operating on. This restriction can be understood by considering the reduction of a virtual hybrid Kernel to a conditional hybrid of concrete hybrids. The conditional hybrid would use the size of the Collection State as its partitioning function – that is, it would partition the State space into subspaces in which the Collection State has a fixed number of children (for example, one subspace might contain only those states where the Collection State has 2 children). The hypothetical conditional hybrid would have a concrete hybrid kernel for each subspace, where that concrete hybrid kernel would have a copy of the virtualized subkernel for each child of the Collection State (see Figure 3-22). In

Appendix A it is shown that such a Kernel structure will have the correct stationary distribution, so long as the virtualized hybrid Kernel cannot change the value of the partition function; that is, cannot change the size of the Collection State.

3.7 Compositionality and state space constraints

One of the primary goals for BLAISE is to allow the construction of probabilistic models and inference algorithms by composing different existing models together. For example, suppose you had data about a number of objects. Each object has several features, and you wish to model each feature using a mixture model, with the constraint that each mixture model uses the same object partitions. How could this be accomplished? To what extent can we reuse our previous Normal mixture model implementation?

BLAISE addresses this issue by supporting State space constraints. These constraints connect disparate portions of the State space, allowing changes in one region of the State space to cause corresponding changes in a different region. Constraints in BLAISE are represented using State→State dependency edges in one of two forms. “In-line” constraints between two States are represented as an edge to the freely changing State from the State reacting to those changes. “Bridged form” constraints introduce a layer of indirection, with a special State whose only function is to reify the constraint. This State has State→State dependency edges to both the freely changing State and the State to be updated by the constraint. Bridged form constraints are useful as a form of “glue”: neither the freely changing State nor the State being updated needed to have their implementation changed; all of the new information lives in the new Constraint State. A bridged form constraint State might not even have an interpretation as a “random variable.” Still, it is considered to be part of the State space because it is an expression of the domain of the State space – in particular, it is a restriction of the valid domain.

For the purposes of Kernels, modifications to the State space as a result of a constraint are considered to be part of the operation that triggered the constraint.

To be more concrete, if a Kernel constructs a Move that modifies one State and a constraint causes a second State to be updated in response, changes to that second state are also considered to be part of the Move constructed by the Kernel. This will become important in section 3.8.

Returning to the motivating example, State space constraints can be used to achieve this multi-feature model, reusing our previous mixture model implementation almost completely. First, we define a new piece of State to represent the definitive partition of objects. This will look like a mixture model, but instead of components with parameters to adjust, the partition will have plain groups, implemented as Collection States. Each object will be reified as a State, with one of the inference goals being to assign the object States to appropriate groups. Next, we add to the model one copy of the mixture model implementation for each feature to be modeled, gathered up using a Collection State. Finally, for each feature we add a constraint State that connects the common object partition to the feature’s mixture model; this constraint State will be responsible for reassigning the feature datapoints to the components that correspond with the object partition’s groups when an object is moved to a new group (see figures 3-23 and 3-24.)

For inference, we can also reuse the existing Kernels, with just a bit of reorganization. A single copy of the Kernels used to reassign datapoints will be used, now targeted at the common object partition. For each feature, we will also have a copy of the component-parameter-adjustment Kernels, gathered using a virtual hybrid Kernel on the Collection State that contains the mixture model States. Finally, we use a concrete hybrid Kernel to compose the reassignment Kernel and the parameter-adjustment Kernel.

It is worth noting that, though some pieces of the SDK graph were combined in new ways, the only new pieces that needed to be implemented were the constraint States required for this particular problem. This is appropriate, insofar as the constraint between mixtures was *exactly* that which was *new* to the problem description. This reflects a general trend with BLAISE: when your modeling language is designed for composition, you only spend your time working on the part of the problem that

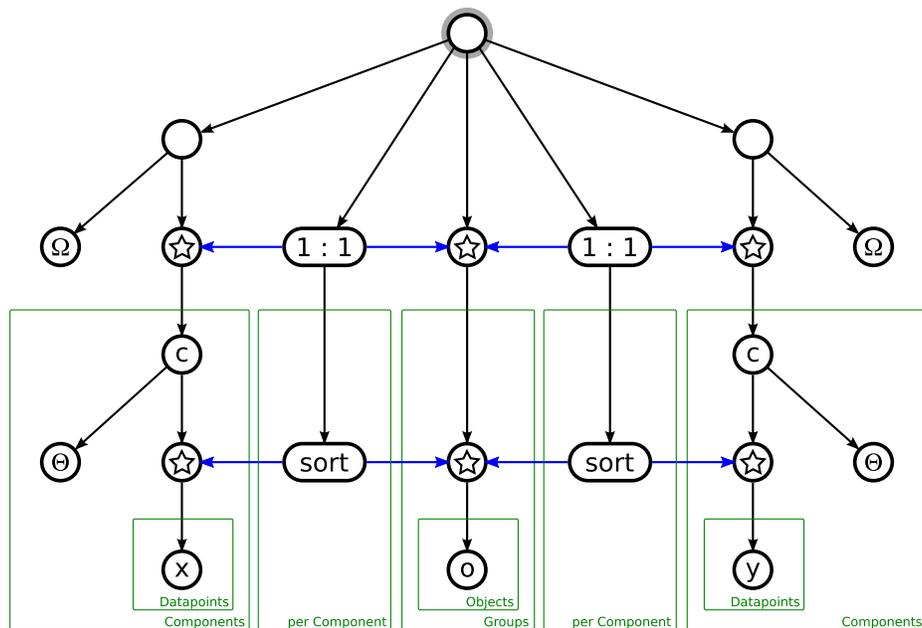


Figure 3-23: Constraint States allow the constraints to be enforced in the state space. This figure shows a two-feature mixture model, written in “bridged form” – that is, using constraint states to bridge between a master partitioning of the datapoints (the center column) and the model for each feature (on the right and left). The feature models are copies of the same mixture model that has been developed throughout this chapter. The States labeled “1 : 1” are Constraint States that maintain a one-to-one correspondence between groups in the master partition and components in the feature models; components are created and destroyed as necessary to maintain this constraint. The States labeled “sort” are constraint States that ensure that when an object is assigned to a component in the master partition, the corresponding datapoint assignment is made in the feature model – that is, that the datapoints in each of the feature models are sorted into the appropriate components. Though both feature models are depicted similarly in this diagram, the joint density and inference procedure for each feature model could be different. For example, the x feature model might be a Gaussian model, while the y feature model was a Beta-Binomial mixture model.

is new. In chapter 6, we will see how a model with even more complicated constraints, an infinite relational model, can be built by reusing the same models we have described here – including inference using exactly the same kernels.

To sum up, constraint States are a general-purpose tool for composing State spaces – they allow State spaces that were designed independently to be “glued” together without modification, and they increase the expressiveness of BLAISE by enabling the values some States to deterministically derived from the values of other States.

3.8 Initialization Kernels

BLAISE is intended for sophisticated modeling. Therefore, let us consider one final extension to our mixture model example: using a Chinese Restaurant Process as a prior for the partition of objects. At first blush, this seems straightforward using our current set of tools: first, add a new Kernel on the common object partition that will create and destroy groups; then, for each feature, add a new constraint State that will ensure that adding a group to the object partition causes a new component to be created for that feature, and likewise destroying a group will destroy the corresponding component. These shouldn’t be too hard to implement: the constraint State is no more complicated than our previous constraint State, and the group-create/destroy Kernel can be implemented easily as a Metropolis-Hastings Kernel with a proposal Kernel that randomly chooses to either create or destroy a group.

Unfortunately, there is a complication. When a new mixture component is created, what value should be assigned to the component parameter Θ ? Moreover, where does the responsibility lie for choosing that value?

One natural approach would be to let the constraint State create mixture components with the parameter Θ set to a pre-determined constant (zero, for example). Notice what this implies about destroying components, however: a proposal to destroy a component can only be accepted by the Metropolis-Hastings Kernel if the inverse move has positive probability. That is, a proposal that destroys a mixture component will only be accepted if there is positive probability of recreating that mixture com-

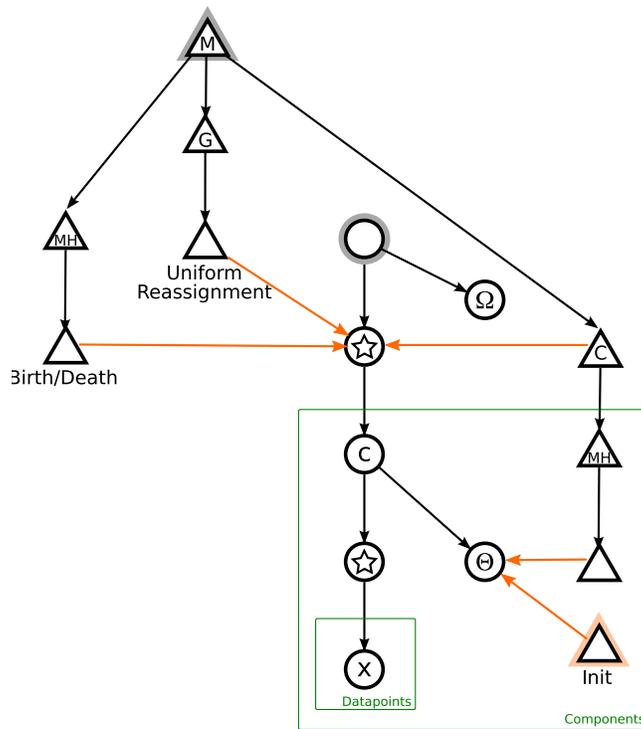


Figure 3-25: When new States are added to the State hierarchy, it is often necessary to sample some aspect of them. For example, if a new component is to be added to a mixture model, any component parameters will need to be sampled. This figure shows a simple mixture model. A Metropolis-Hastings adapted component birth/death Kernel on the left will add and remove components from the model, allowing the number of components to be a target of inference. Each time a component is added, the Initialization Kernel (highlighted with an orange annulus) will be invoked by BLAISE to sample an initial value for Θ .

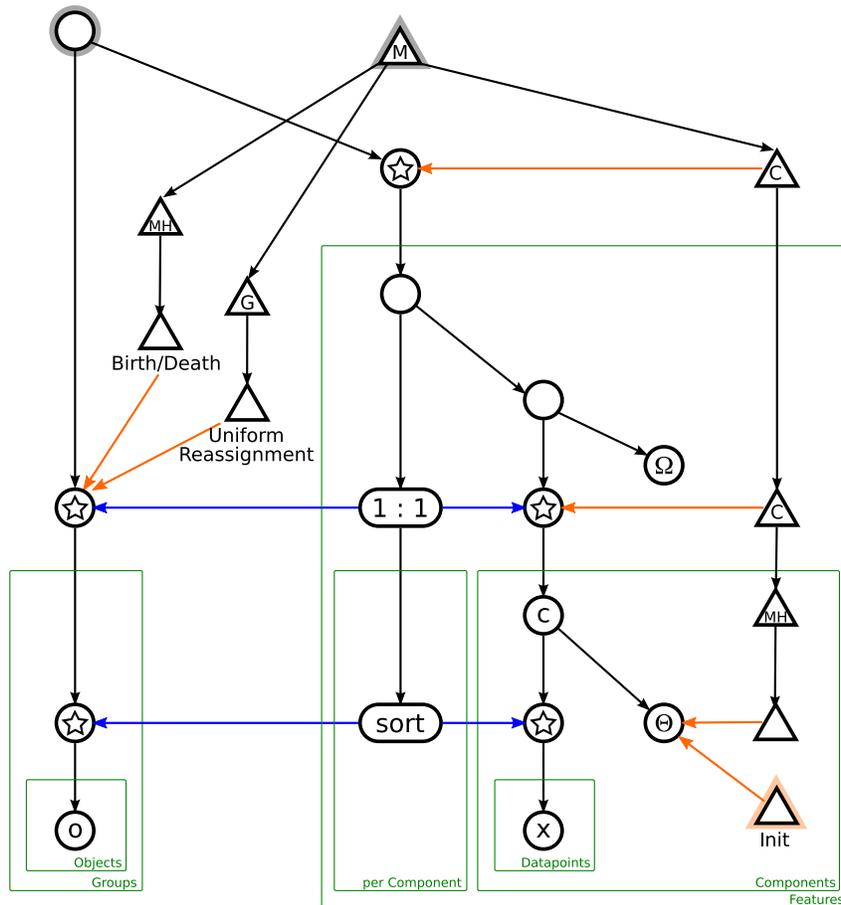


Figure 3-26: Inline mixture models like figure 3-25 could avoid Initialization Kernels by writing a special purpose Birth/Death Kernel that samples the component parameter Θ when a component is created. In contrast, bridged form mixture models like the multiple-feature mixture model shown here require Initialization Kernels in order to support inference on the number of components, because feature model components created by Constraint States need to be initialized.

ponent exactly as it existed before deletion. If mixture components are only created with parameters assigned to pre-determined constants, then mixture components can only be deleted when the parameters equal those pre-determined constants, resulting in extremely poor performance (i.e. slowly mixing Markov chains).

The standard solution to this issue is to sample parameter values from some distribution when mixture components are created. In this way, component deletion proposals can be accepted because there is always a positive probability density of proposing to recreate the component with the same parameter values.

One question remains, though: where does the responsibility for sampling parameter values lie? It is possible that constraint States could sample parameter values; however, the choice of what parameter sampling strategy to use is most naturally an aspect of the inference method description rather than an aspect of the state space description. This suggests that parameter sampling should be a responsibility of the Kernel that causes the components to be created, but this has its own challenges. For example, consider the multi-feature mixture model described in section 3.7. One of the apparent virtues of that model implementation was that additional features could be added without modifying the Kernels that operated on the common object partition – that is, the feature mixture models were thoroughly decoupled from the object partition implementation.

As described previously, Kernels are normally invoked by calling `SAMPLE-NEXT-STATE` on the root of the Kernel hierarchy, with each such call advancing the Markov chain to the next state. `BLAISE` supports decoupled State initialization by introducing a second way for Kernels to be invoked: when new elements of State are added to the State structure and need to have an initial value sampled for them, an *Initialization Kernel* is triggered. Initialization Kernels are bound to specific locations in the State space; for example, one Initialization Kernel might be triggered only by new States being created as a specific Collection State’s children, while a different Initialization Kernel might be responsible for initializing State elsewhere in the State hierarchy.

Initialization Kernels are automatically invoked on pieces of State that needs to be initialized. Returning to the mixture model example, when a new component

is created as part of some Kernel’s SAMPLE-NEXT-STATE operation, the constraint State would give the parameter Θ a dummy value, then add the component to the components Collection State. This would trigger an invocation of the Initialization Kernel’s SAMPLE-NEXT-STATE on the new mixture component, allowing Θ to be initialized. As discussed in section 3.7, modifications to the State space as a result of a constraint are considered to be part of the operation that triggered the constraint; this includes any State initialization done by Initialization Kernels. In the Normal mixture model example, this implies a group birth/death Kernel for the common object partition would produce Moves for which operations such as MOVE-FORWARD-TRANSITION-DENSITY include the probability of any triggered Initialization Kernels sampling the values they did. In other words, there are two ways for one Kernel A to invoke another Kernel B : either A could have B as a child and invoke it directly, or A could cause some change to the State space which triggers an invocation of B as an initialization Kernel; in either case, though, any sampling performed by B on behalf of A will be accounted for by MOVE-FORWARD-TRANSITION-DENSITY, etc. Similar patterns allow the automatic invocation of Initialization Kernels as part of SAMPLE-NEXT-MOVE and ENUMERATE-POSSIBLE-MOVES operations.

Initialization Kernels are also invoked when a previously-initialized State is about to be destroyed. The Initialization Kernel is signaled that this is a destroy operation rather than a construction operation, enabling the Kernel to make the appropriate contributions to the Move (e.g., incorporating into to the Move’s MOVE-REVERSE-TRANSITION-DENSITY value the probability of sampling this exact configuration on a subsequent Initialization).

It is worth noting that hybrid Kernels may also be used to construct Initialization Kernels via composition. For example, one might choose to use a concrete mixture Kernel to randomly choose between two different initialization strategies.

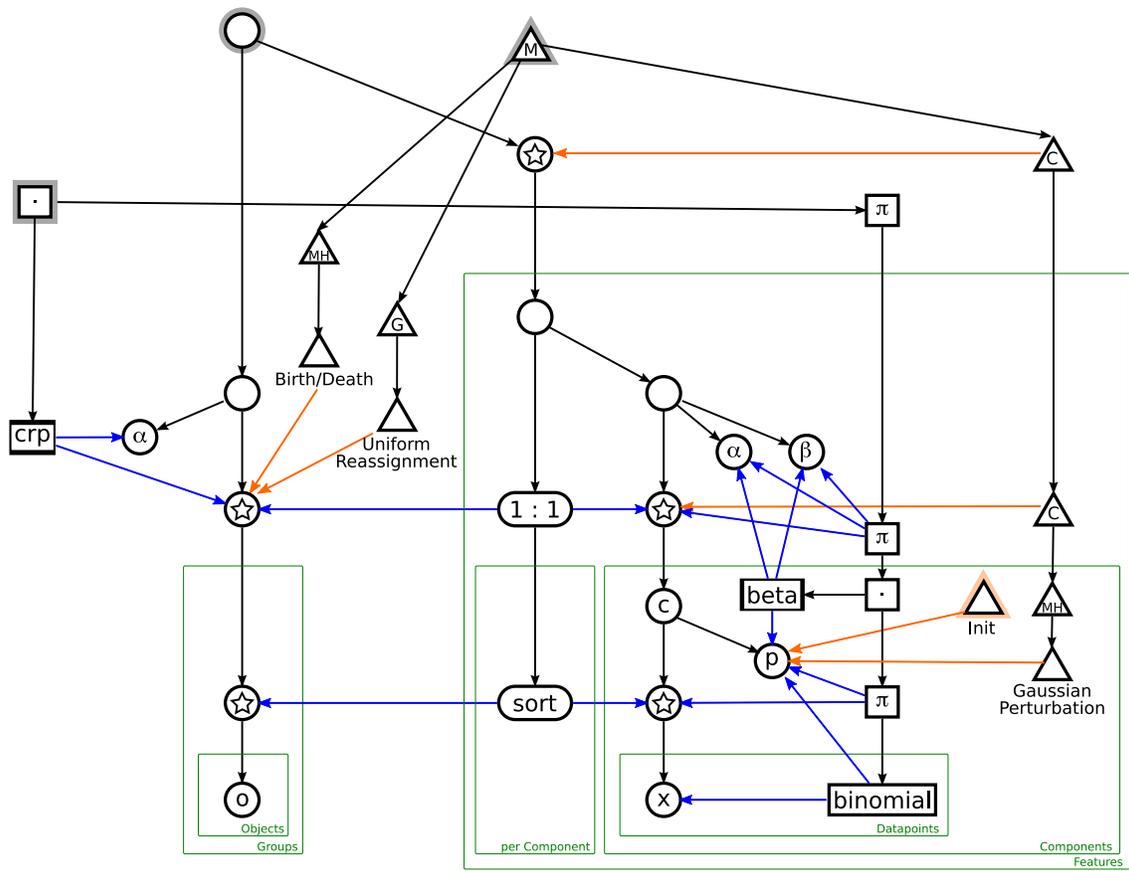


Figure 3-27: This non-parametric multi-feature beta-binomial mixture model is the capstone of this chapter, using almost all the features that have been presented. This model uses a Chinese Restaurant Process (CRP) prior on the partition of objects. As will be seen in chapter 6, the Infinite Relational Model is very similar to this model, and will use exactly the same inference methods.

3.9 Comparison to the Model-View-Controller architecture

A useful analogy may be drawn between the BLAISE’s SDK architecture and the Model-View-Controller (MVC) design pattern [7, 34, 16]. MVC is an architectural design pattern for interactive systems found in a number of the predominant interactive frameworks, including Java Swing [35], Ruby on Rails [59], and Qt (since version 4) [63]. MVC decouples the data representation (the “model”), the presentation of that data via a (graphical) user interface (the “view”), and the interpretation of user input as modifications to the model (the “controller”).

The BLAISE SDK architecture has a similar structure (see figure 3-28b). States fill the role of the MVC model as the data representation abstraction. Densities are then similar to views: they provide a particular interpretation of the data. Whereas MVC Views are (typically) used to reduce complex data to a 2-dimensional visualization, Densities reduce the complex data to a 1-dimensional continuous quantity: the value of the joint density on the State space. Finally, Kernels are analogous to MVC controllers, with BLAISE’s random bit stream filling the role of MVC’s user input. The MVC controller interprets user input in relation to the view in order to make appropriate changes to the model; these changes will then be reflected in the view. Likewise, a BLAISE Kernel interprets random bits from the bit stream in relation to the Density in order to make appropriate changes to the model; these changes will then be reflected in the Density. Note that it is natural to bring the MVC user and the BLAISE random bit stream into correspondence, because both are the primary sources of non-determinism in their respective application domains.

3.10 Comparison to reactive programming languages

BLAISE’s SDK architecture also has interesting similarities with reactive programming languages (for example, FrTime [11]). Reactive programming languages seek to model the flow of information as data changes over time. A statement such as “ $a = b + c$ ”

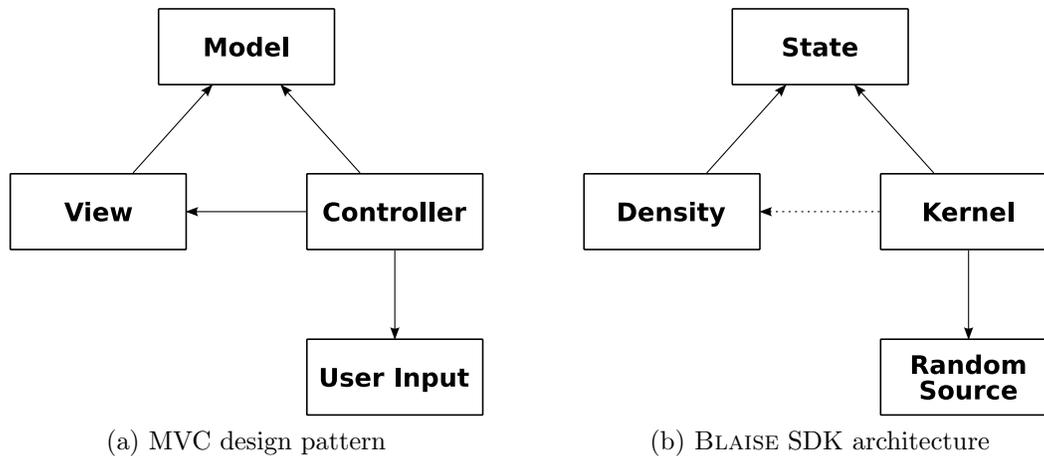


Figure 3-28: The Model-View-Controller architectural design pattern and the BLAISE SDK architecture are analogous: Models and State represent data, Views and Densities provide an interpretation of that data, and non-determinism (either from user input or random bit sources) are interpreted by Controllers and Kernels. In this diagram, the dependency of Kernels on Densities is drawn as a dashed arrow to emphasize the narrowness of this communication channel (a single number encoding the evaluation of the joint density).

in a non-reactive language would be interpreted as “compute the sum of the current values of b and c , and assign that value to a ,” even if b or c were to have its value changed, a would still maintain the same value (the original sum). In a reactive language, a similar statement would have a more constraint-like interpretation, i.e. ensure that a always has the value $b+c$. In such a language, a *dataflow* would connect a to b and c so that the the value of a will *react* to changes in the value of b or c .

BLAISE Densities are much like expressions in reactive programming languages: they define a computation on a set of values; those values may change over time, and the Densities are expected to reflect such changes. Drawing a correspondence to the example above, a Density might fill the role of a , a State the role of b , and another Density the role of c ; filling the role of addition would be some function f of the form $a = f(b, density(c))$. Reactive programming languages are generally concerned with directed acyclic dataflows. Similarly, BLAISE Densities are required to be trees (a subclass of acyclic graphs); considering Densities and the States they depend upon results in a singly-rooted directed acyclic graph¹⁰.

¹⁰Density “dataflows” are singly rooted, even though general reactive programming data-flows

Unlike most other languages, reactive programming languages typically reify and store the result of intermediate computations, even if these values are never directly accessed later; this reification and storage is required to determine how changes should propagate through the dataflow. As will be described in chapter 5, the BLAISE virtual machine uses a similar strategy by memoizing the value of all Density evaluations. Also like many reactive programming language implementations, the BLAISE VM uses an invalidation/lazy-evaluation scheme to maintain efficiency by only evaluating the necessary portions of large dataflows/Density hierarchies.

3.11 Comparison to Infinite Contingent Bayes Nets

Infinite Contingent Bayes Nets (ICBNs) [44] are another approach to representing graphical models in which there may be an unbounded number of objects. ICBNs are much like standard Bayes Nets, except that edges may be labeled with conditions (boolean expressions on the other variables in the graph) that must be satisfied for the edge to be active (that is, to have an effect on the joint distribution). ICBNs underlie the implementation of BLOG [45], a first order probabilistic modeling language.

BLAISE SDK models bear a relation to ICBNs. Unlike BLAISE models, ICBNs have fixed topology; however, the set of active edges is dynamic and changes over the course of inference. This is similar to how the structure of the Density hierarchy in BLAISE changes over the course of inference – the set of currently existing Densities and their connections to the State hierarchy is analogous to the set of active edges in an ICBN. BLAISE SDK models provide advantages over ICBNs, such as avoiding index variables and supporting undirected models.

are not. This reflects the fact that BLAISE models are only interested in a single joint probability density defined over the state space, whereas in general reactive programming, there may be multiple quantities of interest.

3.12 Discussion

In this chapter, I supported my thesis statement by inventing the BLAISE State–Density–Kernel (SDK) graphical modeling language. In the BLAISE SDK language, the state space is represented using a tree of States. Distinguishing features of the State representation include the ease and explicitness of composition, the encoding of information in the mutable topology of the state space, the ability to include unknown numbers of objects (using Collection States), and the incorporation of state space constraints (using Constraint States).

Densities were then presented as a means of encoding a joint probability density that decomposes naturally over the state space. The BLAISE density model is distinguished by providing explicit control over the composition of Densities, which enables the density structure to reflect unknown numbers of objects (using Associated Collection Densities) and which provides for non-linear composition strategies (such as tempering Densities).

Next, this chapter presented BLAISE’s unique approach to modeling inference by adding Kernel nodes to BLAISE graphical models. Kernel composition was described, including an interpretation of traditional composition strategies, such as concrete cycle and mixture kernels, in terms of the BLAISE framework, as well as the development of novel composition strategies such as conditional hybrid Kernels and virtual cycle and mixture Kernels. In addition, the Metropolis-Hastings algorithm and Gibbs sampling for value-enumerable States were reinterpreted as generic BLAISE Kernels. Initialization Kernels were also invented as a means of decoupling the logic for initializing new States from the logic for determining when new State should be created.

In this chapter, I also put forward the hypothesis that the probability density evaluations required for inference can generally be couched in terms of the joint probability density – that is, evaluations of the root Density. This hypothesis is supported by the implementations of the Metropolis-Hastings and Gibbs Kernels outlined in this chapter, and will be further supported in chapter 4.

Finally, throughout this chapter I detailed how BLAISE is designed from the

ground up for composability, including describing the composition strategies for States, Densities, and Kernels, showing how useful models can be built up piece-by-piece and by recycling existing models. It is particularly striking that in many cases, the modeler can continue to use exactly the same inference Kernels even as the model grows to significant complexity. The soundness of the design was also justified by drawing an analogy between the BLAISE SDK probabilistic inference architecture and the well-established Model-View-Controller interactive system architecture, as well as an analogy between the BLAISE States and Densities and the dataflows in reactive programming systems.



Chapter 4

BLAISE Transformations

My thesis is that a framework for probabilistic inference can be designed that enables efficient composition of both models and inference procedures, that is suited to the representational needs of emerging classes of probabilistic models, and that supports recent advances in inference.

In this chapter, I support this thesis by highlighting how several recent advances in inference are supported by interpreting them as graph transformations in the SDK language.

By the end of this chapter, you will be able to use BLAISE transformations to automatically convert a normal BLAISE SDK model to a version that uses simulated annealing, parallel tempering, or particle filtering. You will be able to transform a model for a single mixture component into a variety of complete mixture models, or to integrate out a variable in a conjugate model.

4.1 An introduction to BLAISE transformations

Chapter 3 introduced the BLAISE SDK graphical modeling framework, focusing on the patterns of composition enabled by the BLAISE abstractions. In this chapter, I will explore another virtue of the the SDK modeling language: its ability to be programmatically manipulated. Such manipulations take the form of transformations on the SDK graph.

SDK transformations have many uses. One might transform an SDK graph in order to extend the probabilistic model; for example, to transform a mixture component into a mixture model. One could also transform an SDK graph in order to enhance inference. The Metropolis-Hastings transform, introduced in section 3.4.2 and used to correct the stationary distribution of a kernel, is such an inference enhancement transform. There are many other examples, however, including transformations for simulated annealing, parallel tempering, and particle filtering. Finally, BLAISE transformations may be useful to make an existing SDK graph more efficient: for example, by rewriting portions of the model to exploit conjugacy.

Different kinds of transformations are likely to be used in different ways. For example, model extension transforms such as the mixture model transform will likely only be useful when specifically requested by a user during the model creation process. Efficiency transforms, however, could be deployed automatically as part of an optimizing compilation stage that is invoked just before inference begins. Inference enhancement transformations could be used in either fashion, depending on the transformation; for example, Metropolis-Hastings transforms are likely to be under direct user control, while the parallel tempering transformation might be invoked by an optimizing compiler. Still, even if parallel tempering were only available as a user-controlled operation, the ability to convert an existing inference algorithm to a parallel tempered version of the same algorithm using one line of code (or maybe even just a point-and-click operation) is of incredible value to the modeler, insofar as it reduces both development *and* computation time.

The transformations presented here are far from an exhaustive set; rather, the goal of this chapter is to provide enough familiarity with BLAISE transformations that the reader can get a feeling for their utility and for how even advanced inference techniques, such as parallel tempering, can be made accessible if the modeling language uses the appropriate abstractions.

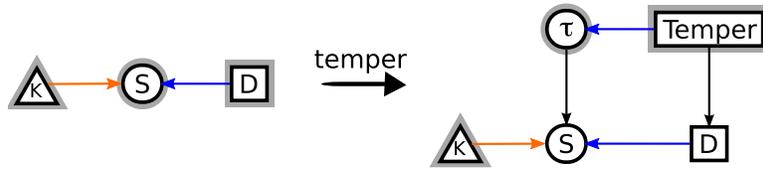


Figure 4-1: The *temper* transformation is used to change the temperature at which a model is evaluated (section 4.2.1). Tempering is an important component of simulated annealing (figure 4-2) and parallel tempering (figure 4-3).

4.2 Tempered models

The shape of a joint density landscape can be modified by a process called *tempering*, wherein a joint density is evaluated at a specific “temperature” τ by raising the joint density to the power of $1/\tau$. As τ approaches infinity, (i.e. when the temperature is very hot), the distribution becomes relatively flat, assigning nearly equal probability to all events in the support of the original joint density. Such hot distributions are useful because they are easier to explore. For example, it is easier to produce a Metropolis-Hastings Kernel that mixes well for a hotter distribution, because it is less likely that proposals will be made to low-probability regions of the state space. As τ approaches zero (i.e. when the temperature is very cold), the distribution becomes very “peaky,” assigning all the probability mass to the maximum *a posteriori* (MAP) state. Such cold distributions are useful because MCMC samples from a cold distribution are more likely to produce samples at or near the MAP value. Note that the original joint density is recovered when $\tau = 1$.

As described in section 3.3, BLAISE models can have Densities that introduce non-linearities, and therefore can represent inference techniques based on tempering as transformations of the BLAISE SDK model. The section introduces the TEMPER transform to temper a BLAISE model, then builds upon this transform to implement simulated annealing and parallel tempering transforms.

4.2.1 The temper transform

The TEMPER transform extends an existing SDK model to produce a tempered version:

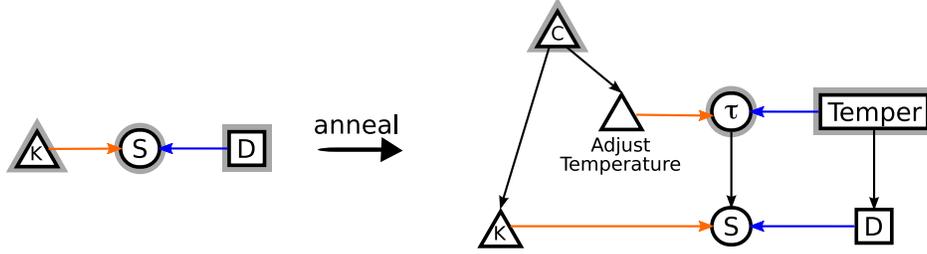


Figure 4-2: Simulated annealing in BLAISE is implemented by the ANNEAL transform (section 4.2.2). This transformation uses the TEMPER transform (figure 4-1) to create a version of the model with an adjustable temperature τ . A concrete cycle Kernel is used to adjust the temperature on every iteration of inference.

$$\langle S, D, K \rangle, \tau \xrightarrow{\text{temper}} \langle S_\tau, D_{\text{temper}}, K \rangle$$

where S is a State, D is a Density, and K is a Kernel (see figure 4-1). Under this transformation, S_τ will have S as a child and will contain the variable for the temperature τ . D_{temper} will have D as a child density and a reference to S_τ , with the density being evaluated as $\text{density}(D_{\text{temper}}) = \text{density}(D)^{1/\tau}$. The Kernel K is unaffected.

The TEMPER transform is relatively simple, but the next two sections will show how simulated annealing and parallel tempering can be implemented using this transform as a subroutine.

4.2.2 Simulated Annealing

Simulated annealing [17, 32] is a temperature-based inference enhancement scheme in which a model is tempered to a hot temperature, then slowly cooled to either $\tau = 1$ (to produce samples from the untempered joint distribution), or to a τ near 0 (to produce a MAP estimate).

Simulated annealing can be implemented as a BLAISE transformation that takes a standard BLAISE SDK model and an annealing schedule (i.e., a description of how aggressively to reduce the temperature), and produces an annealed SDK model:

$$\langle S, D, K \rangle, \text{schedule} \xrightarrow{\text{anneal}} \langle S_\tau, D_{\text{anneal}}, K_{\text{anneal}} \rangle$$

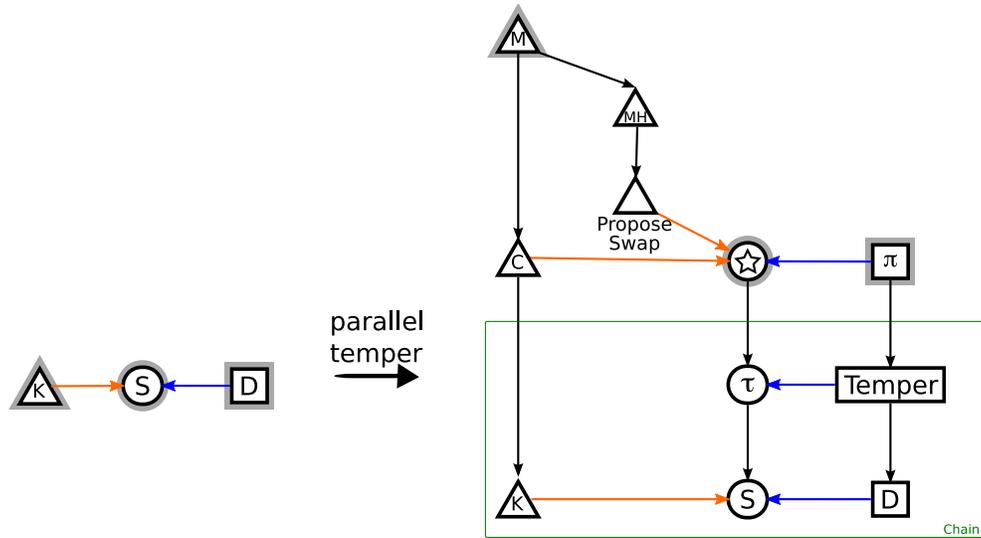


Figure 4-3: Parallel tempering in BLAISE is implemented by the *parallel temper* transform (section 4.2.3). This transformation uses the *temper* transform (figure 4-1) to create multiple copies of the original model, each running at a different temperature. A virtual cycle Kernel is used to update each copy of the model using K . Occasionally, the mixture Kernel will choose to apply the swap-chains Kernel instead of the advancement Kernel; the swap-chains Kernel proposes swapping a pair of chains with adjacent temperatures.

(see figure 4-2). Under this transformation, S_τ and D_{anneal} are tempered versions of S and D (produced using the *TEMPER* transform):

$$\langle S, D, K \rangle, \tau_1 \xrightarrow{\text{temper}} \langle S_\tau, D_{anneal}, K \rangle$$

where τ_1 is the initial temperature for the annealing schedule. The Kernel K_{anneal} is a concrete cycle Kernel with two children: K and a Kernel that makes a deterministic update to τ according to the annealing schedule.

4.2.3 Parallel Tempering

Parallel tempering [18, 15] is another temperature-based inference enhancement scheme. In parallel tempering, a number of copies of a model are produced and each is tempered to a different temperature; each copy represents a (nearly) independent Markov chain. Inference is performed on each of the chains independently and in parallel, except that occasionally chain inference is paused and there is an opportunity to

swap the state between two chains with similar temperatures. Swaps are accepted or rejected such that each chain is guaranteed to produce samples from its tempered distribution. Specifically, if a chain has $\tau = 1$, it is guaranteed to produce samples from the untempered joint density, just as standard MCMC would; the goal, though, is to produce samples more efficiently than standard MCMC (by converging faster and mixing better).

Parallel tempering can be viewed as a generic method for providing intelligent proposals for a Metropolis-Hastings kernel. For example, consider a two-chain parallel-tempered system, where the cold chain has $\tau_{cold} = 1$ and the hot chain has $\tau_{hot} > 1$. The hot chain has a flatter, easier-to-search distribution; therefore, we expect MCMC on the hot chain to converge faster and mix more effectively than MCMC on the cold chain. Assuming the temperature difference between the two chains isn't too large, however, the hot chain's distribution is still similar to that of the cold chain. Considering a swap is much like using the hotter distribution to propose the next value for the cold chain, and the τ_{hot} parameter can be adjusted to make it sufficiently likely that swap proposals will be accepted while maximizing the ease with which the hot chain mixes.

Parallel tempering can be implemented as a BLAISE transformation

$$\langle S, D, K \rangle, \text{schedule}, \text{period} \xrightarrow{\text{parallel temper}} \langle S_{pt}, D_{pt}, K_{pt} \rangle,$$

where *period* is the expected number of steps to advance each of the parallel chains between attempts to swap chains (see figure 4-3). Under this transformation, S_{pt} is a Collection State having one child state, $ChainState_i$, for each temperature in $\text{schedule} = \tau_1, \dots, \tau_n$, and D_{pt} is an Associative Collection Density with one child Density, $ChainDensity_i$, for each temperature. $ChainState_i$ and $ChainDensity_i$ are then created using the tempering transform

$$\langle S, D, K \rangle, \tau_i \xrightarrow{\text{temper}} \langle ChainState_i, ChainDensity_i, K \rangle.$$

The kernel K_{pt} is the composition of two simpler kernels: $K_{advance}$ and K_{swap} .



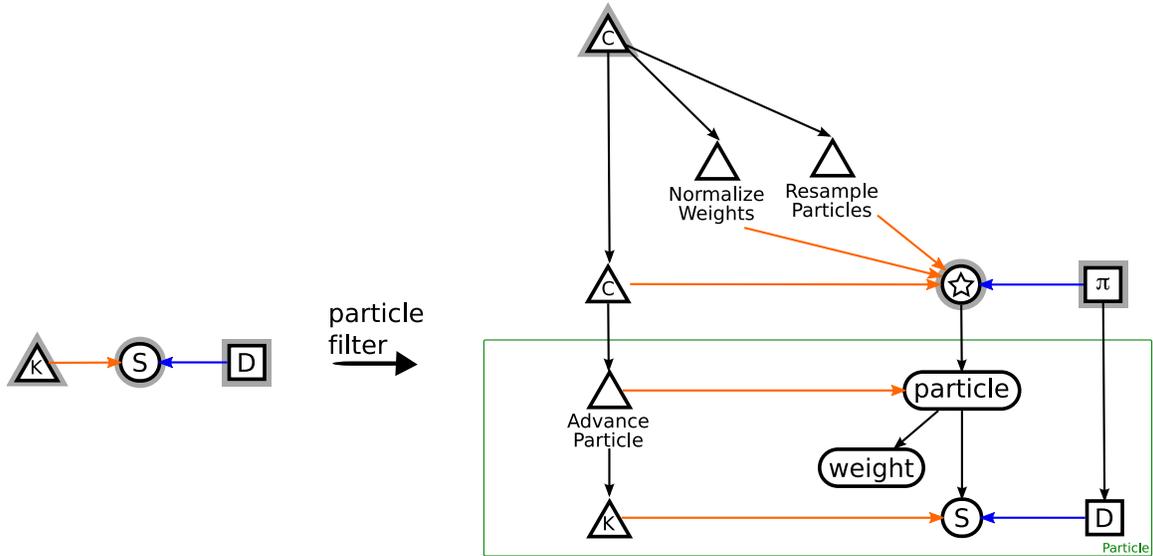


Figure 4-4: Particle filtering in BLAISE is implemented by the *particle filter* transform (section 4.3). The input to the transformation is an empty initial state S and a density D for that state, together with a Kernel K that makes sequential extensions to the state (e.g. adds variables for next time step of a dynamic Bayes net, sampling values for those variables). This transformation makes several copies of the original model to use as particles; each copied State is embedded in an *particle* State that also contains the particle weight. A cycle hybrid kernel $K_{advance}$ is used to update each copy of the model and its associated weight. Next, $K_{normalize}$ renormalizes all the weights, and finally $K_{resample}$ optionally resamples the particles. A cycle hybrid kernel K_{pf} orchestrates $K_{advance}$, $K_{normalize}$, and $K_{resample}$.

$K_{advance}$ is used to advance each of the chains one step. It is implemented as a virtual cycle Kernel, virtualizing K over the collection of chains. K_{swap} attempts to swap chains with adjacent temperatures. It is constructed as a Metropolis-Hastings Kernel, where the proposal Kernel selects an index $i \in [1, n - 1]$ and proposes swapping the S part of $ChainState_i$ and $ChainState_{i+1}$. This proposal is then stochastically accepted according to the standard Metropolis-Hastings acceptance ratio. The hybrid Kernel K_{pt} is a concrete mixture Kernel that applies K_{swap} with probability $1/(period + 1)$; otherwise it applies $K_{advance}$.

4.3 Particle Filtering

Particle filtering, also known as sequential Monte Carlo, is a population-based variant of importance sampling. Several samples, called particles, are built up in parallel (i.e. the same variable will be sampled in all the particles, then the next variable will be sampled in all the particles, and so on). As in standard importance sampling, a weight is associated with each particle to encode how well the sampling process for the particle matches the interest distribution – for example, how well the sampling process is able to account for the observed evidence. Before moving on to sample the next variable, the particle filter may optionally resample the population of particles by sampling from the current set of particles in proportion to their weights. This process tends to discard particles that explain the data poorly, while generating new particles in those regions of the state space that explain the data well.

Although particle filtering is not typically thought of as a Markov chain Monte Carlo method, it is still a Monte Carlo method that can be implemented in a Markov chain. As will be seen, particle filtering inference can even be constructed atop the same SDK framework described throughout this thesis for MCMC, despite not being analyzable as an MCMC method (but see [46]).

In BLAISE, particle filtering is implemented as an SDK transformation

$$\langle S_0, D, K_+ \rangle, \#particles \xrightarrow{\text{particlefilter}} \langle S_{pf}, D_{pf}, K_{pf} \rangle,$$

where S_0 is an “empty” state (i.e. no variables sampled yet), D is the interest distribution, and K_+ is a Kernel that makes sequential extensions to S_0 (see figure 4-4). For example, if the model is a dynamic system, then S_0 might be the model for just the initial conditions of the system, and K_+ might add the next unincorporated time step to the S by adding the appropriate variables for that time step and sampling values for those variables.

S_{pf} is a Collection State that has $\#particles$ children $S_{particle}$, where each $S_{particle}$ is a particle, implemented as a composite State containing a copy of $S_{particle-model}$ and a real-valued State $S_{particle-weight}$ encoding the particle’s weight. Initially, all of

the particles have a weight of $1/\#particles$.

D_{pf} is an Associative Collection Density containing a copy of D for each S_i .

K_{pf} is a concrete cycle Kernel with three children, $K_{advance}$, $K_{normalize}$, and $K_{resample}$. $K_{advance}$ is a virtual cycle Kernel that applies a Kernel $K_{advance-particle}$ to each particle S_i in S_{pf} . $K_{advance-particle}$ uses K_+ to sample a move $S_t \xrightarrow{K_+} S_{t+1}$. It then advances the particle's model State to reflect S_{t+1} and updates the particle's weight using the update rule

$$S_{particle-weight} \leftarrow S_{particle-weight} \cdot \frac{density_{root}(S_{t+1})}{density_{root}(S_t)} \cdot \frac{1}{MOVE-FORWARD-TRANSITION-DENSITY(S_t \xrightarrow{K_+} S_{t+1})}$$

$K_{normalize}$ normalizes all of the particle weights by computing the sum of all the particles weights, then dividing each particle's weight by that sum.

Finally, $K_{resample}$ optionally resamples the particles by drawing a set of $\#particles$ independent and identically distributed new particle states, sampled with replacement from the existing set of particles and weighted by the particle's weight. The newly sampled particles' weights are all reset to $1/\#particles$.

4.4 Hybrid Algorithms

Because everything in BLAISE is implemented in terms of the same three foundational abstractions, it is easy to create hybrid inference procedures. For example, in BLAISE, hybridizing reversible jump MCMC with other inference methods such as parallel tempering is simply a matter of applying the parallel tempering transform to a model using the reversible jump features of the standard Metropolis-Hastings Kernel (see figure 4-5). More sophisticated hybrids are also possible, such as combining particle filtering with Markov chain Monte Carlo methods (see figure 4-6) [19]. In BLAISE, this can be achieved by using either a cycle hybrid or mixture hybrid kernel to interleave applications of the particle filtering kernel K_{pf} with applications of a

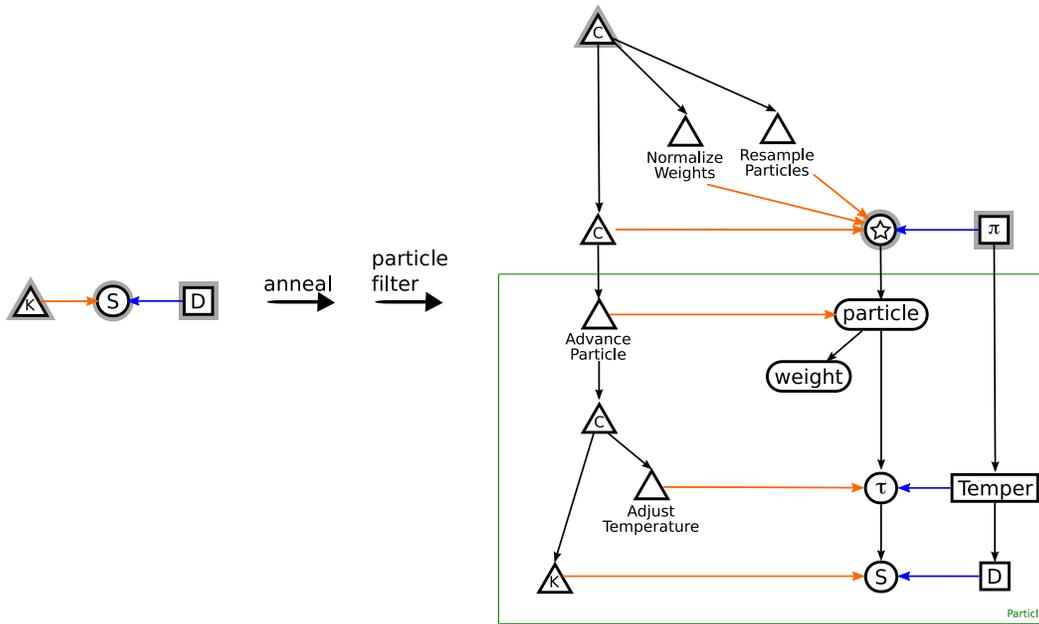


Figure 4-7: BLAISE allows the concatenation of transformations. Here, the simulated annealing transform has been applied to a model, after which the particle filter transformation was applied. (Note: analyzing this hybrid inference method for correctness and performance is outside the scope of this thesis; the method is included here as an example of BLAISE’s potential to support such research.)

accept my wager, you might want to make several inferences using a mixture model, where each mixture component corresponds to a coin, and each datapoint encodes the coin flips resulting from a single draw from the bag. You will need to infer both the coin weights (component parameters) and which coin was drawn from the bag for each datapoint.

4.5.1 Parametric mixture models with fixed size and fixed weights

If it was known ahead of time how many coins were in the bag, as well as how likely it was that a particular coin would be drawn from the bag, then you might consider using a parametric mixture model with a fixed number of components and known mixture weights. The mixture model transformation given these assumptions is:



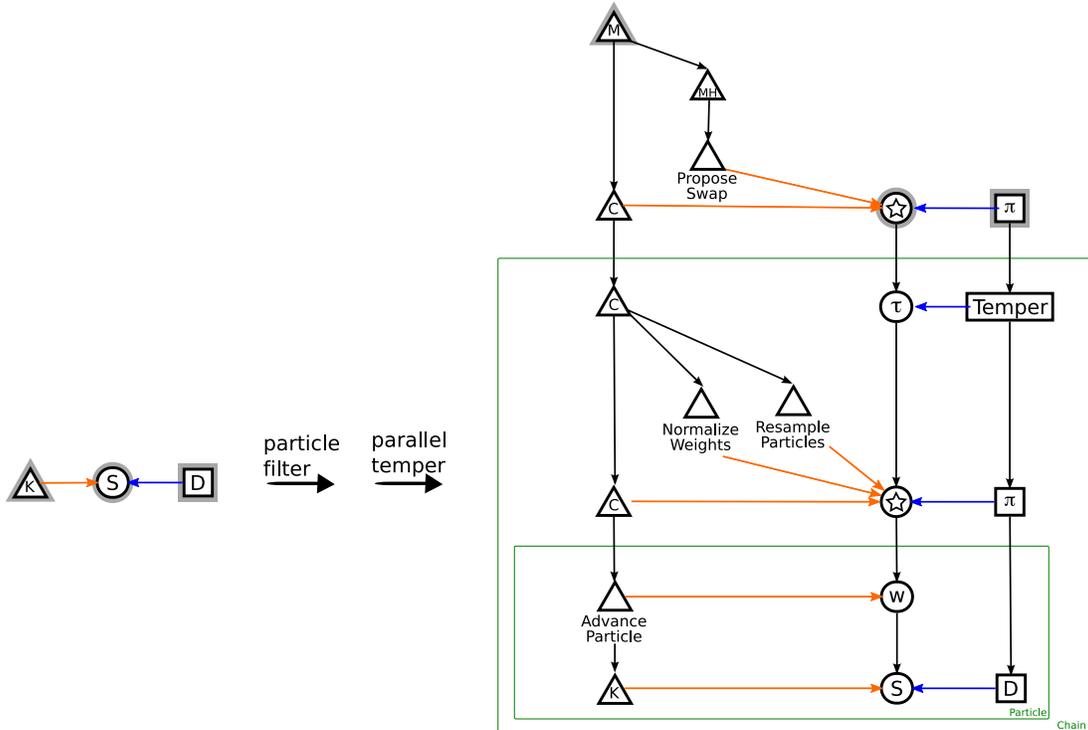


Figure 4-8: BLAISE allows the concatenation of transformations. Here, the particle filter transformation has been applied to a model, after which the parallel tempering transformation was applied, resulting in a parallel tempered particle filter. (Note: analyzing this hybrid inference method for correctness and performance is outside the scope of this thesis; the method is included here as an example of BLAISE’s potential to support such research.)

$$\langle S, D, K \rangle, \#components, weights \xrightarrow[\text{fixed size and weights}]{\text{parametric mixture}} \langle S_{mix}, D_{mix}, K_{mix} \rangle,$$

where S is a mixture component implemented as a Collection State that has the datapoints as children, $weights$ is a vector of $\#components$ component weights summing to 1, and K is an inference kernel on the parameters of S . The transform first extends S with a weight by creating a new State S_{comp} that contains S and a real-valued State S_w . S_{mix} is then a Collection State containing $\#components$ copies of S_{comp} , with each weight from $weights$ being assigned to one component’s S_w State. D_{mix} is an Associated Collection Density with one copy of the component Density D_{comp} for each mixture component in S_{mix} , where D_{comp} is a Multiplicative Density

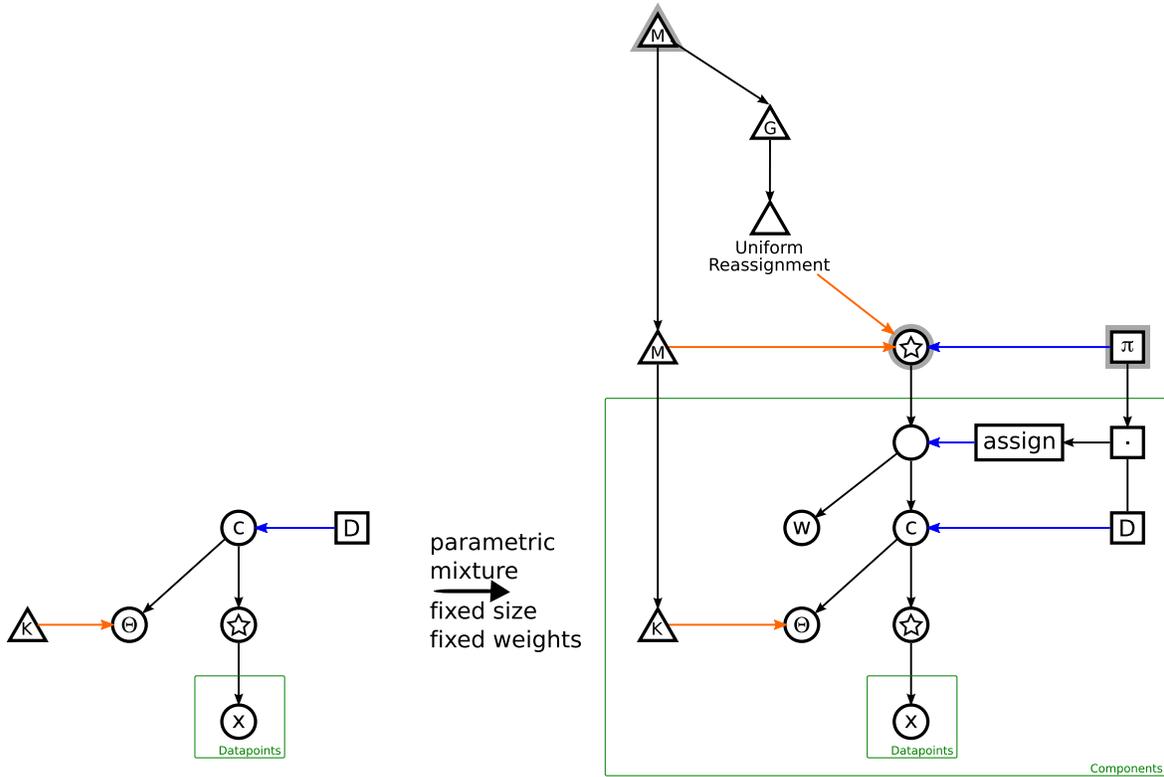


Figure 4-9: A mixture model transformation that transforms a model for a mixture component into a complete parametric mixture model with fixed size and fixed component weights.

containing a copy of D (to evaluate the probability of the model parameters and the likelihood of the data assigned to that component) and also containing a Density D_{assign} to compute how likely it is that the datapoints were assigned to this component (i.e. $S_w^{\#dp}$, where $\#dp$ is the number of datapoints assigned to this component). K_{mix} is a concrete hybrid Kernel that composes a datapoint reassignment Kernel (for example, a Metropolis-Hastings Kernel that proposes moving a datapoint from one mixture component to another) and a virtual hybrid Kernel that applies the component-parameter-inference kernel K to each of the components.

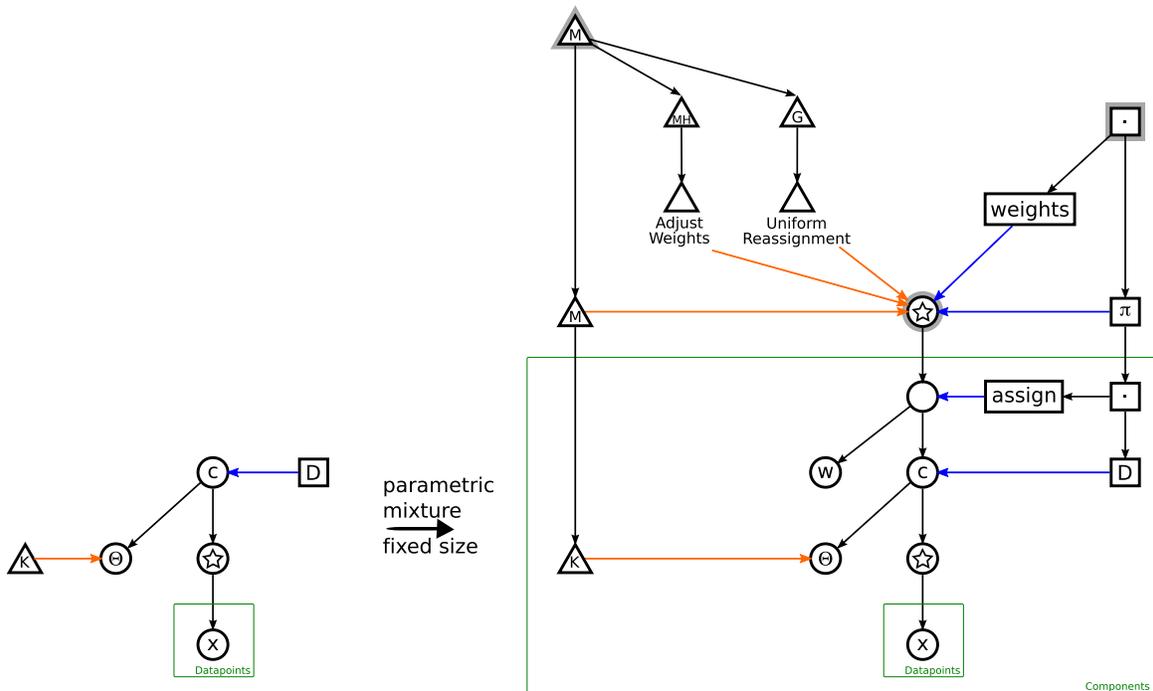


Figure 4-10: A mixture model transformation that transforms a model for a mixture component into a complete parametric mixture model with fixed size, but with component weights to be inferred.

4.5.2 Parametric mixture models with fixed size and unknown weights

What if some coins were more likely to be drawn from the bag than others, but it was not known ahead of time what the probability of drawing a particular coin was? This situation can be modeling by relaxing the fixed-weight assumption, as in the fixed-size unknown-weights transform:

$$\langle S, D, K \rangle, \#components \xrightarrow[\text{fixed size}]{\text{parametric mixture}} \langle S_{mix}, D_{mix}, K_{mix} \rangle,$$

that first applies the fixed-weight parametric mixture transform

$$\langle S, D, K \rangle, \#components, \overrightarrow{1/\#components} \xrightarrow[\text{fixed size and weights}]{\text{parametric mixture}} \langle S_{mix}, D_{fixed}, K_{fixed} \rangle,$$

where $\overrightarrow{1/\#components}$ is a vector of length $\#components$ where each entry's value is $\frac{1}{\#components}$. The transformed model is then extended by creating a new Multiplicative

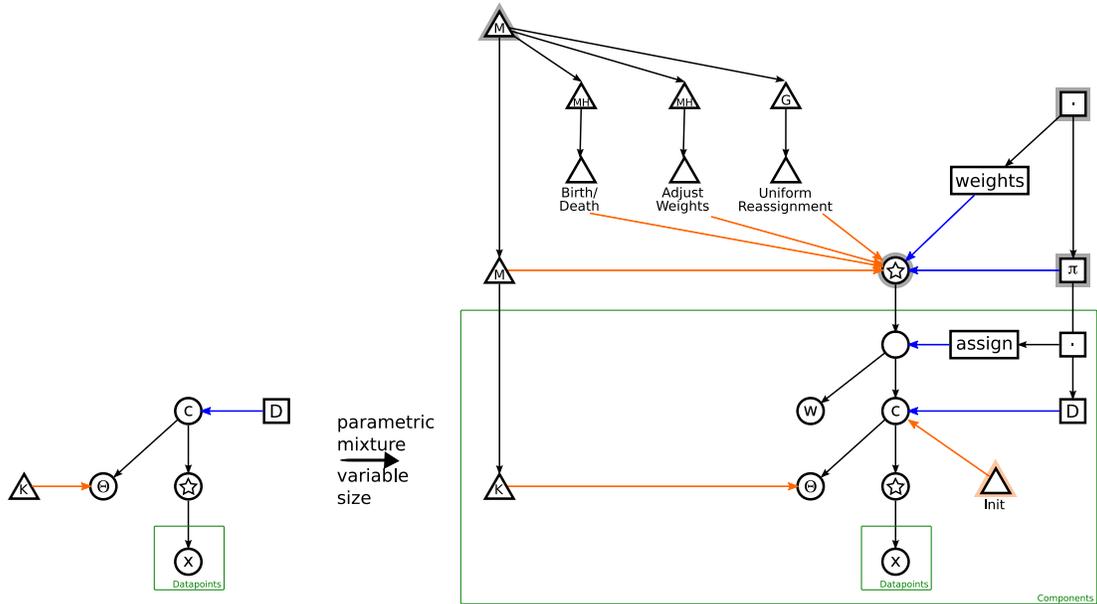


Figure 4-11: A mixture model transformation that transforms a model for a mixture component into a complete parametric mixture model, with the number of components to be inferred.

Density D_{mix} containing D_{fixed} and a prior distribution over the mixture weights – say, a symmetric Dirichlet distribution. Finally, K_{mix} is a concrete hybrid kernel composing K_{fixed} and a Kernel $K_{weights}$ that adjusts the mixture weights.

4.5.3 Parametric mixture models of unknown size

Suppose even the number of coins were in the bag was unknown; you would have to infer this while playing the game. This situation could be modeled using a prior distribution on the number of components, as in the variable-size parametric mixture transform:

$$\langle S, D, K \rangle, p(\#components), K_{init} \xrightarrow[\text{variable size}]{\text{parametric mixture}} \langle S_{mix}, D_{mix}, K_{mix} \rangle,$$

where $p(\#components)$ is a prior on the number of components and K_{init} is an initialization kernel that can sample parameter values for a component. This transform



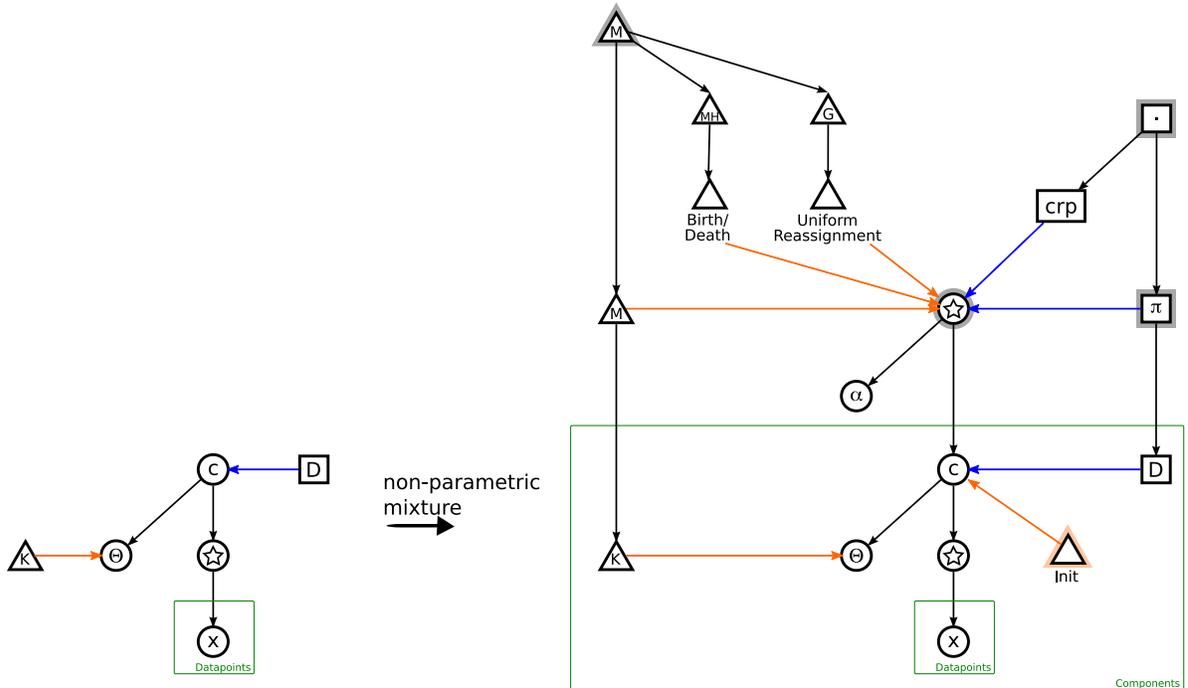


Figure 4-12: A mixture model transformation that transforms a model for a mixture component into a complete non-parametric mixture model.

would first apply the fixed-size parametric mixture transform

$$\langle S, D, K \rangle, 1 \xrightarrow[\text{fixed size}]{\text{parametric mixture}} \langle S_{mix}, D_{fixed}, K_{fixed} \rangle.$$

The transformed model is extended by creating a Multiplicative Density D_{mix} containing D_{fixed} and a Density $D_{\#components}$ that evaluates $p(\#components)$ on the number of components in S_{mix} . A new concrete hybrid Kernel is also created that contains K_{fixed} and a new Kernel $K_{birth/death}$ that can create and destroy mixture components. Finally, K_{init} is attached to the model such that components created (or destroyed) by $K_{birth/death}$ will have their parameters properly initialized (or de-initialized).

4.5.4 Non-parametric mixture models

Non-parametric mixture models are similar to variable size parametric mixture models. However, instead of using component weights and a prior over the number of components, non-parametric mixture models use a single prior distribution over the

partitioning of the datapoints into groups (components), such as a Chinese Restaurant Process (CRP) distribution. The non-parametric mixture transformation can be specified as:

$$\langle S, D, K \rangle, \alpha, K_{init} \xrightarrow{\text{non-parametric mixture}} \langle S_{mix}, D_{mix}, K_{mix} \rangle,$$

where α is the parameter of the CRP prior on partitions. S_{mix} is then a Collection State containing $\#components$ copies of S as well as a real-valued State S_α to hold the value α . D_{mix} is a Multiplicative Density containing D_{CRP} that evaluates the CRP prior and an Associated Collection Density with one copy of the component Density D for each mixture component in S_{mix} . K_{mix} is a concrete hybrid Kernel that composes:

- a virtual hybrid Kernel that applies the component-parameter-inference kernel K to each of the components.
- a datapoint reassignment Kernel $K_{reassign}$ (for example, a Metropolis-Hastings Kernel that proposes moving a datapoint from one mixture component to another)
- a component creation and destruction kernel $K_{birth/death}$, as in the variable size parametric mixture transform.

Finally, K_{init} would be attached to the model such that components created (or destroyed) by $K_{birth/death}$ will have their parameters properly initialized (or de-initialized).

4.5.5 Bridged-form mixture models

The mixture model transformations in this section have all produced *inline* mixture models, in which the partitioning of objects (datapoints) into groups is implemented by the same structure that implements the mixture components themselves. For each inline mixture model transform, an analogous transform for *bridged form* mixture models, as described in section 3.7, could also be written.

gate prior for the BERNOULLI distribution, it is possible to analytically compute the posterior distribution on θ :

$$p(\theta|x_1 \dots x_N) = \frac{p(\theta|\alpha, \beta) \prod_i p(x_i|\theta)}{\int p(\theta|\alpha, \beta) \prod_i p(x_i|\theta) d\theta} = \text{BETA}(\theta; \alpha + \#heads, \beta + (N - \#heads))$$

where $\#heads$ is the number of heads in $x_1 \dots x_N$.

Furthermore, it is also possible to analytically compute the predictive density of a conjugate model $p(x_{N+1}|x_1 \dots x_N, \alpha, \beta)$; that is, given some data $x_1 \dots x_N$, what is the probability of the next flip being heads? For example, for the Beta-Bernoulli model, the predictive distribution is

$$p(x_{N+1}|x_1 \dots x_N, \alpha, \beta) = \frac{B(\alpha + \#heads_{1..N}, \beta + (N - \#heads_{1..N}))}{B(\alpha + \#heads_{1..N+1}, \beta + (N + 1 - \#heads_{1..N+1}))}$$

where $B(x, y) = \int_0^1 t^{x-1} (1-t)^{y-1} dt = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$ is the Beta function.

Note that the posterior distribution and the predictive distribution depend only on summary information about the datapoints: the total number of datapoints, and the number that were heads. In general, for conjugate models this type of summary information, called *sufficient statistics*, is all that is required to fully track the posterior distribution and predictive distribution. Furthermore, the sufficient statistics are easy to update as datapoints are added to and removed from the component. By maintaining a running product of the predictive distribution for each new datapoint that is added to the model, it is also possible to incrementally compute the marginal likelihood of all the data (i.e., the joint density of the component).

$$p(x_1 \dots x_N | \alpha, \beta) = \prod_{i=1}^N p(x_i | x_1 \dots x_{i-1}, \alpha, \beta)$$

A conjugacy transformation can be used to perform the “integrating out” operation in a BLAISE conjugate model (see figure 4-13). For example, a Beta-Bernoulli conjugate transform would eliminate the State representing θ , the Beta Density connecting S_α , S_β , and S_θ , and the Bernoulli Densities connecting θ to each of the

datapoints. The transform would then add a Beta-Bernoulli conjugate Density that uses sufficient statistics of the data to compute the marginal likelihood.

4.7 Discussion

In this chapter, I supported my thesis by describing several classes of BLAISE SDK transformations in the SDK language, including efficiency-enhancing transformations and model extension (i.e. composition-focused) transformations, as well as transformations that interpret several recent advances in inference as automated changes to the graph structure. The transformations provide significant power to the modeler. Whereas techniques such as parallel tempering are often considered to be too complicated to implement for most projects, BLAISE transformations make the difference between an untempered model and a parallel tempered model a matter of a single line of code, or a single click if a GUI were being used. Moreover, BLAISE transformations suggest how one might construct an optimizing compiler for probabilistic inference methods by automatically identifying opportunities to apply transformations to an SDK model.

Chapter 5

The BLAISE Virtual Machine

My thesis is that a framework for probabilistic inference can be designed that enables efficient composition of both models and inference procedures, that is suited to the representational needs of emerging classes of probabilistic models, and that supports recent advances in inference.

In this chapter, I support this thesis by describing the BLAISE Virtual Machine, a software system that can efficiently execute the inferences represented by BLAISE SDK graphs.

5.1 An introduction to the BLAISE Virtual Machine

Chapter 3 introduced the BLAISE State–Density–Kernel (SDK) graphical modeling language and identified several ways in which the SDK language is more flexible than traditional graphical modeling languages. Chapter 4 built upon this flexibility, demonstrating how a range of inference enhancements, model extensions, and model simplifications can be implemented as generic transformations of SDK graphs. In this chapter I introduce the BLAISE Virtual Machine (VM), a software framework that executes the stochastic processes described by SDK graphs on common off-the-shelf computers. I focus on those aspects of the BLAISE VM that support the flexibility of

the SDK language while maintaining efficient operation.

The BLAISE VM is implemented in Java. Each State, Density, and Kernel in a BLAISE model is represented as a Java object. BLAISE provides abstract base classes for States, Densities, and Kernels; each of these classes extends a common graph node base class, Node, that provides support for assembling the SDK graphical model from the individual State, Density, and Kernel objects. Node provides support for directed graph semantics, and allows edges to be efficiently traversed in either direction. Node also allows specific edges to be efficiently located based on a number of criteria, including the role the the edge plays (e.g. State→State versus State→Density), incidence (i.e. incoming versus outgoing), and user-specified tags (for example, a State containing two children, one representing a random variable α and one representing a random variable β , might tag its outgoing State→State edges “alpha” and “beta,” respectively.) The Node abstraction also provides facility for passing messages across across incoming edges (e.g. from a Node to its “parents”). Messages can be selectively propagated across only those incoming edges that have certain roles in the graph (e.g. A message might propagate across only State→Density or Density→Density edges). The specific messages used by the BLAISE VM will be described shortly.

The VM supplies abstract base classes for each of the central representations (State, Density, Kernel) as well as standard modeling components for each of those representations. Specifically, the VM provides:

- a State abstract base class, along with States for primitive variables (e.g. Integer State, Real State, Boolean State, etc.) and Collection States. The VM also makes it easy to create composite States.
- a Density abstract base class, along with Densities for common probability distributions (e.g. Gaussian, Poisson, Chinese Restaurant Process, etc.), Densities for conjugate models (e.g. a Beta-Binomial Density as described in section 4.6), and Multiplicative and Associated Collection Densities. The VM also makes it easy to create composite Densities.
- a Kernel abstract base class, along with Concrete Hybrid Kernels (i.e. Concrete Mixture Kernels, Concrete Cycle Kernels, Concrete Conditional Kernels),

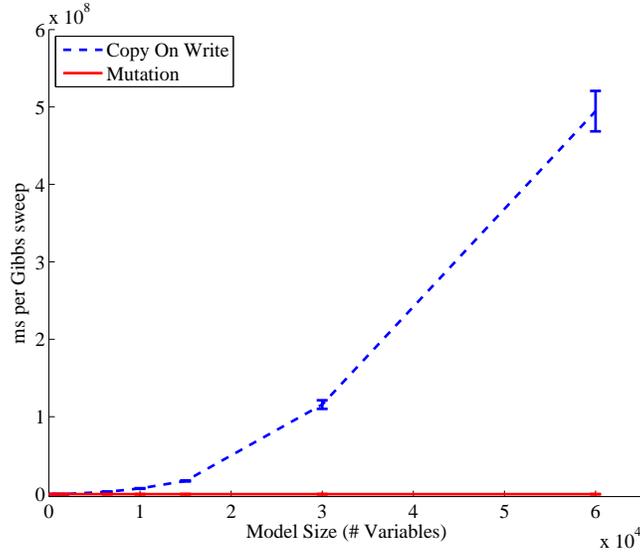
Virtual Hybrid Kernels (i.e. Virtual Mixture Kernels, Virtual Cycle Kernels), Kernels for specifying the piece of a state space that another Kernel should operate on (called “Let Kernels”), Kernels for Metropolis-Hastings and enumerative Gibbs Sampling (as described in section 3.4.2), and Kernels for performing simple inference on primitive variables (for example, the Gaussian Perturbation Kernel used for Metropolis-Hastings on real-valued States in section 3.4.2).

5.2 States are mutated in-place

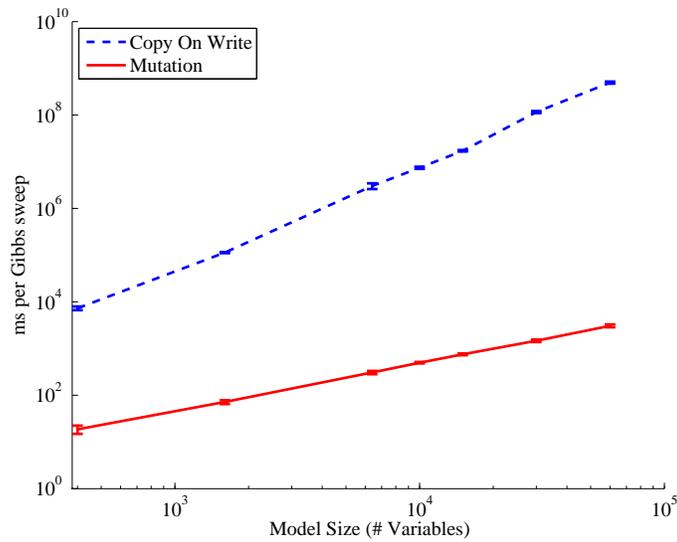
Moving around the state space is the most central operation in BLAISE, and therefore must be as efficient as possible. For this reason, States are mutated in place rather than copied. For example, a Kernel’s `SAMPLE-NEXT-STATE` is an operation that takes a state S_t and samples a next state S_* by mutating the S_t to become S_* . In-place mutation is more efficient both in terms of space (no need for multiple copies of the State hierarchy to be held in memory) and in terms of time (no need to spend time copying the State hierarchy.) Figure 5-1 demonstrates the performance benefits of in-place mutation.

5.3 Density evaluations are memoized

As State and Density structures grow more complex, it will often be the case that changes to a small piece of the State space will only cause the value of a small number of the Densities to change. For example, changing the parameters in one component of a mixture model will not affect the Densities attached to any of the other components. The BLAISE VM therefore memoizes Density evaluations; whenever a Density is evaluated, the value is cached in the Density. Whenever a State changes value or structure, the BLAISE VM ensures that all dependent Densities have their memos flushed. This is achieved by having the changing State emit a message which propagates up `State→Density` and `State→State` edges; whenever a Density receives such a message, the Density flushes its cache. The Density also emits a message that



(a) Linear



(b) Log-Log

Figure 5-1: The BLAISE Virtual Machine uses in-place mutation for efficiency. This figure shows the runtime data for a family of different-sized models, contrasting in-place mutation and the alternative copy-on-write strategy, plotted on a linear scale in (a) and a log-log scale in 5-1b. The models are using models: simple lattice-structured factor graphs with boolean variables (see section 6.5). The model size was varied between 400 variables (20x20 graph) and 60,000 variables (200x300 graph). The data plotted here is the average time cost of a Gibbs sampling sweep through the entire model, averaged over 10 runs, each with 20 sweeps. Copy-on-write cost was calculated by running the mutation-based inference, eliminating any time spent on mutation and transaction bookkeeping, and adding the empirical cost of a model copy every time a model copy was needed. Note that the copy-on-write strategy gets exponentially more costly as the model size increases; this is because every site in the Gibbs sweep requires the model to be copied.

propagates up Density→Density edges so that all parent Densities flush their cache as well. The next time $density(D_{root})$ is evaluated, all these Densities will have their value recomputed using the new State values.

This memoization is critical to BLAISE’s efficiency and eliminates a lot of complications when designing algorithms. For example, consider computing the Metropolis-Hastings acceptance ratio. One of the terms in this ratio, $\frac{p(s_*)}{p(s_i)}$ involves comparing the joint density of two states: the proposed state and the current state. When crafting an efficient algorithm by hand, a practitioner will often symbolically manipulate the joint densities to cancel terms that are known not to change, so that time is not spent computing these terms. These considerations must be made separately in each M-H kernel, because each kernel will make changes that affect different terms in the joint density. Any change to the structure of the joint density also requires reconsidering which terms will change. Considering both the large number of times when these considerations must be made, and the fact that this tends to be an error-prone process, the result is in an extremely brittle system.

The BLAISE VM relies on the automated memoization of Densities instead of manual cancellation of terms. In a BLAISE M-H Kernel, the Densities whose caches are invalidated by a move $S_T \xrightarrow{K} S_*$ are exactly the Densities whose values change because of the move; that is, the same set of Densities that would remain after the manual cancellation of terms described above. The BLAISE VM therefore achieves similar performance to the symbolic approach, while remaining automated and robust to the modification of M-H proposal Kernels or changes to the joint density landscape. Figure 5-2 demonstrates the execution efficiency gained by density memoization.

5.4 Tree-structured Transactional Caching for States and Densities

Memoization conserves a lot of computation, but there is still significant opportunity for wasted computation whenever changes to the State space are “undone.” For



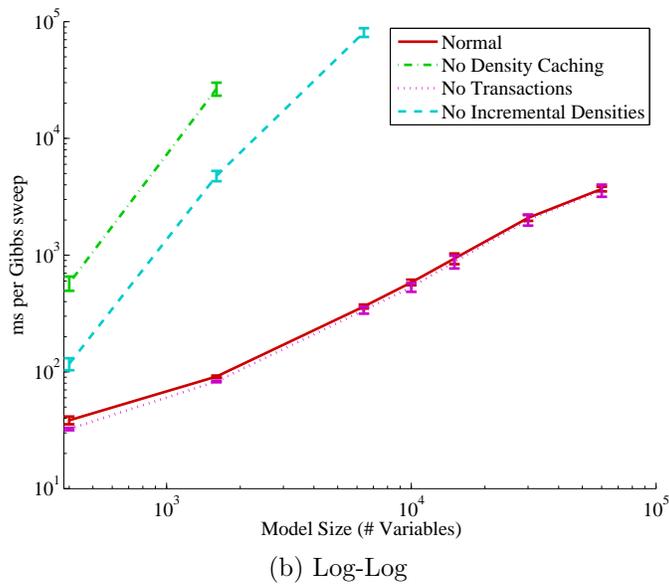
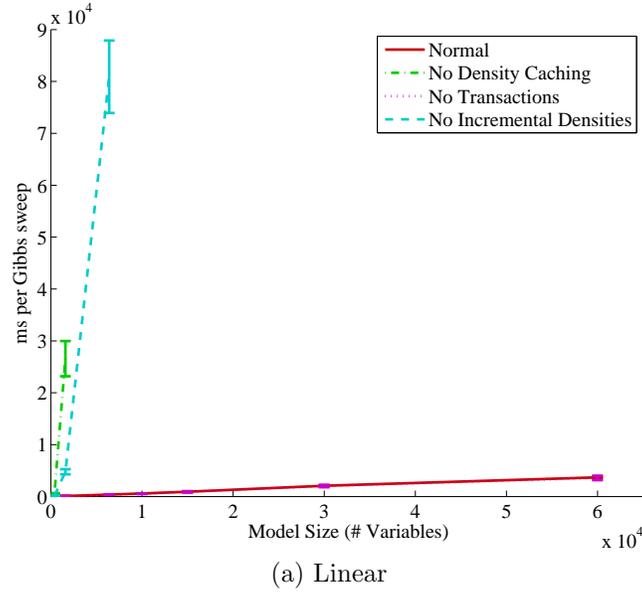
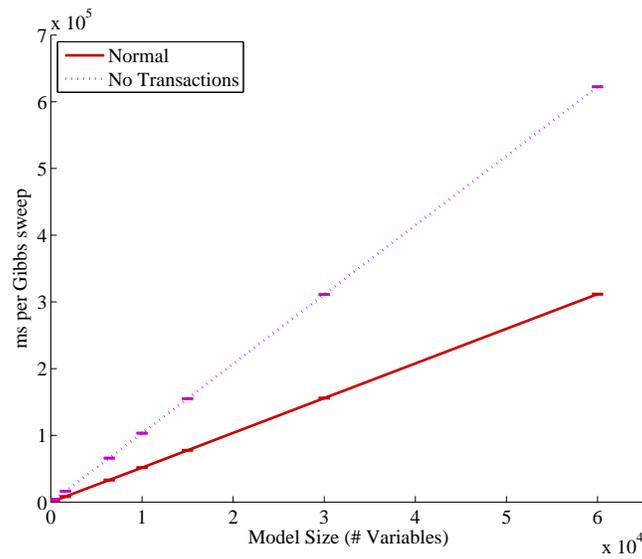
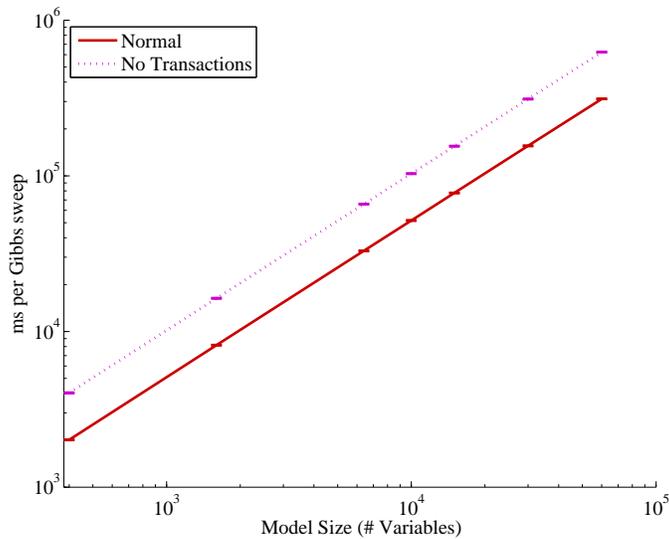


Figure 5-2: The BLAISE Virtual Machine uses a variety of memoization techniques in order to maintain efficiency while supporting the generality of the BLAISE SDK modeling language. This figure shows the effects of disabling various elements of the BLAISE VM’s memoization support, using the same models as in figure 5-1. “Normal” indicates BLAISE’s normal operation, using memoized densities managed by a hierarchical transaction system, and using incremental update logic for Multiplicative Collection Densities and conjugate Densities. “No Density Caching” disables density memoization (including incremental updates), “No Transactions” disables the hierarchical transaction system, and “No Incremental Densities” disables just the incremental update logic. Inference runs were limited to 1 hour (3,600,000 ms). Note that the “No Transaction” line tracks the “Normal” line. This is because the factors in an Ising model are extremely quick to compute; figure 5-3 will expand on this.



(a) Linear



(b) Log-Log

Figure 5-3: In the experiment in figure 5-2, the “No Transaction” line tracks the “Normal” line. This is because the factors in an Ising model are extremely quick to compute, as each factor can only take on two values that are known at model creation time (see section 6.5 for more details). If each factor takes just one additional millisecond to compute, as in this experiment, the advantage of “Normal” over “No Transaction” becomes apparent. Many standard probability distributions require non-trivial computation to evaluate. In some models, such as the Generative Vision model in section 6.1, evaluating a single Density could easily take tens to hundreds of milliseconds.

example, consider a Metropolis-Hastings Kernel which proposes the move $S_t \xrightarrow{K} S_*$. To evaluate the acceptance ratio, the Kernel first evaluates $density(D_{root})$ while the State hierarchy is configured to S_t , then updates the State hierarchy to reflect S_* and evaluates $density(D_{root})$ again. Suppose the Kernel rejects the proposal, and reverts the State space back to S_t – if the next Kernel also needs to evaluate $density(D_{root})$, should it have to re-evaluate all the Densities that are dependent on the States that the Metropolis-Hastings Kernel touched, even though the State hierarchy is back in S_t , and $density(D_{root})$ was computed for this configuration just moments ago?

The BLAISE VM eliminates this wasted computation by using a transaction system to manage States and Densities. The VM’s transaction management system allows a Kernel to begin a State-Density transaction, make changes to the State space which result in changes to the Density space, and then roll back the transaction to efficiently return the State and Density hierarchies to their original configuration. A rolled-back transactions can also be re-applied, which will efficiently put the State and Density hierarchies in the configuration they were in before the transaction was rolled-back. Transactions can be committed (making the applied configuration permanent) or aborted (making the rolled-back configuration permanent).

The BLAISE VM can nest transactions in other transactions. For example, suppose one Kernel begins a transaction, then invokes a second Kernel to do some work on the State space. The second Kernel is permitted to begin its own transaction, and this transaction would be nested inside the first Kernel’s transaction. If the second Kernel commits its transaction, any mutations performed as part of that transaction will be absorbed by the outer Kernel’s transaction.

A stack of nested transactions is sufficient to support a Metropolis-Hastings Kernel, but what about an enumerative Gibbs Kernel? Such a Kernel evaluates $density(D_{root})$ on many candidate states before sampling just one to be the actual next state. The transaction manager also supports this interaction pattern by extending the stack of transactions to a tree of transactions; that is, it is possible to begin a transaction, roll it back, begin a new transaction (parallel to the first one), roll that back too, then re-apply and commit either of the transactions. With such a system, even an

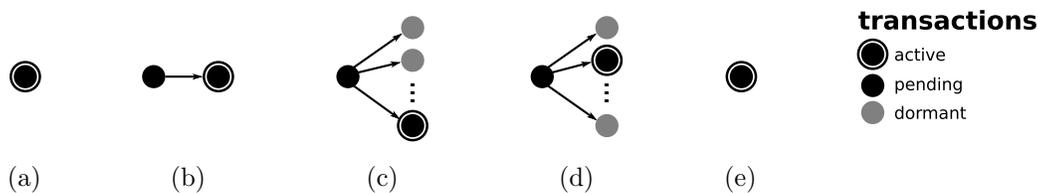


Figure 5-4: BLAISE uses a tree-structured transactional caching system to enhance performance. This figure shows how the caching system supports Metropolis-Hastings kernels and enumerative Gibbs sampling over a variable with a finite, discrete domain. (a) The caching system always has exactly one *active* transaction: the transaction with which Density changes will be associated. (b) Before doing any speculative work, such as considering a Metropolis-Hastings proposal, a new transaction opened. This transaction becomes active, while the previous transaction enters the *pending* state: the changes associated with it are reflected in the current State, but new changes will be associated with a different transaction (the active one). After evaluating the proposal, a Metropolis-Hastings kernel would either commit the new transaction (causing its changes to be incorporated into the parent transaction), or it would abort (discarding changes). (c) Enumerative Gibbs kernels must evaluate many possible moves; a new transaction is opened for each of these. Transactions that are siblings to the active transaction are *dormant*: their changes are not reflected in the current State. (d) Once a move has been chosen, the transaction associated with that move is reactivated and committed, simultaneously aborting all its sibling transactions. (e) Both Metropolis-Hastings kernels and enumerative Gibbs kernels return the transaction tree to its original structure by the end of their operation.

enumerative Gibbs Kernel does only as much computation as is strictly necessary. (See figures 5-4, 5-2, and 5-3.)

5.5 Incremental Updates for Efficient Collection Densities

Because Multiplicative Densities are such a common Density composition tool in BLAISE, special care was taken in the Virtual Machine to ensure that these Densities are efficient.

Consider a Multiplicative Collection Density with some number of child Densities. Suppose a Kernel modifies a State that affects just one of those child Densities. The child density attached to this State will have its memoized value cleared; all the other child Densities will retain any memoized values they had. The Multiplicative Collec-

tion Density’s memoized value will also have its memoized value cleared, reflecting the fact that it needs to be recomputed because one of its children has changed value. A naïve implementation of the Multiplicative Collection Density would just recompute the product of its children’s values from scratch; only the changed child would have to do any work to determine its value, but just asking each child for its value means that the Collection Density’s operation would take time linear in the number of children. If the number of children is large, this can be a significant waste of effort, and such situations are not uncommon. For example, Multiplicative Densities could be used to support mixture models that may have large numbers of components, such as the Infinite Relational Model (section 6.3), or to support large graphical models (section 6.5). Introducing an $O(\#children)$ slowdown is therefore highly undesirable.

Instead, the Multiplicative Collection Densities in the BLAISE VM are implemented using an incremental update mechanism that reduces the $O(\#children)$ operation to an $O(k)$ operation, where $k \leq \#children$ is the number of recently invalidated child densities. The Multiplicative Density partitions its child densities into two disjoint sets: C_{used} and $C_{pending}$. It also maintains an internal cache of the value $density_{used} = \prod_{C_i \in C_{used}} density(C_i)$. When a child density $C_i \in C_{used}$ is invalidated, the cache is updated by the rule $density_{used} \leftarrow \frac{density_{used}}{density(C_i)}$, where $density(C_i)$ represents the density before taking into consideration the invalidation-causing change; C_i is also moved from C_{used} to $C_{pending}$. Then $density(d)$ can be evaluated by performing the update $density_{used} \leftarrow density_{used} \prod_{C_i \in C_{pending}} density(C_i)$, moving all densities from $C_{pending}$ to C_{used} , and returning $density_{used}$. The size of the set $C_{pending}$ is k because it contains exactly those child Densities that have been modified since the last evaluation of the Multiplicative Density; thus evaluating Density in $C_{pending}$ is an $O(k)$ operation. (See figure 5-2.)

Other Densities can benefit from similar incremental-update mechanisms. For example, conjugate model Densities (see section 4.6) such as the Beta-Binomial Density use incremental updates to maintain the sufficient statistics for the conjugate model and to maintain a running product of predictive densities so that the marginal likelihood can be evaluated efficiently.

5.6 State, Density, and Kernel responses to State modifications

Modifications to the State hierarchy, especially changes to the structure of the hierarchy, can require many sympathetic changes elsewhere in the SDK model: Densities may evaluate to a different value, Constraint States may need to alter a constrained piece of the State hierarchy, Associated Collection Densities may need to construct a new child Density or remove an existing child Density, and virtual hybrid Kernels may now have a different number of States to cycle or mix over. I have already explained how Densities values are kept up to date, but how do the rest of these updates happen efficiently?

Section 5.3 described how States emit messages when changing value or structure. Constraint States maintain their constraints efficiently by responding to these messages. Whenever such a message is received, the Constraint state responds by making sympathetic changes to other portions of the State space to which the Constraint State is connected. Note that this process can cascade – one change can cause a Constraint State to make another change, which in turn causes a more distant Constraint State to make a change, and so on.

Similarly, an Associated Collection Density responds to messages from its associated Collection State; messages indicating that a child State was added cause the Associated Collection Density to construct a new Density and attach it to the new child State, whereas messages indicating a child State was removed cause the corresponding child Density to be removed from the Associated Collection Density.

Kernels maintain no state whatsoever (unlike States, which are clearly stateful, as well as Densities, which have internal state to enable caching). Furthermore, in the BLAISE VM representation of SDK models, Kernel→State edges aren't even represented; instead, the State that a Kernel should operate on is passed to the Kernel as a parameter (see figure 5-5 for more details). Therefore, Virtual Hybrid Kernels do not require a separate copy of the virtualized child Kernel for each child of the Collection State that the hybrid kernel will mix or cycle over. Instead, the Hybrid

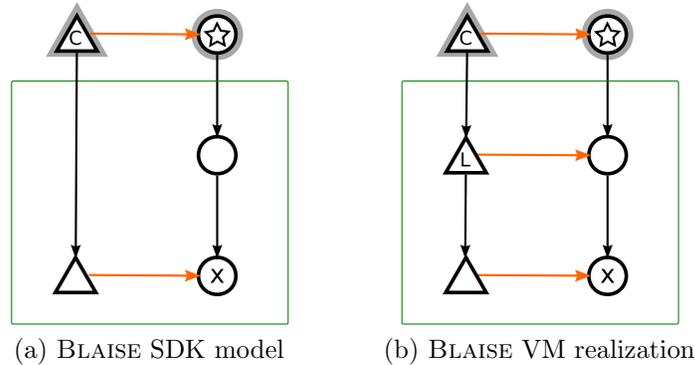


Figure 5-5: Kernels are realized in the BLAISE VM using a method-call-with-arguments semantics, so that virtual hybrid Kernels may be implemented more efficiently. Every composite Kernel in the BLAISE VM has a contract specifying what parameters (States) will be passed to its child Kernels when they are invoked, given the parameters that were passed to the composite Kernel. For example, a virtual cycle Kernel expects a Collection State as its parameter, and will invoke its child Kernel once for each State in the collection, passing the Collection State’s child State as the argument to the child Kernel. Sometimes, the BLAISE SDK model dictates that a child Kernel should act on a descendent of the child State, rather than the child State itself, as in figure (a). The BLAISE VM realizes such models using a “Let” Kernel, indicated with an “L” in figure (b). A Let Kernel is a composite Kernel that selects a specific descendent of its argument State and passes that descendent to its child Kernel.

Kernel simply determines which children are currently contained in the Collection State that it is passed at invocation time. For example, a Virtual Cycle Kernel would be passed a Collection State S_* and would find the Collection State’s children $S_1 \dots S_N$. The Virtual Cycle Kernel would then invoke its virtualized child Kernel N times, each time passing as an argument a different child State (i.e., a different element in the set $\{S_1 \dots S_N\}$). Because Kernels are invoked using an argument-passing paradigm rather than actually having edges connecting to the State hierarchy, the Kernel structure does not need to change over the course of inference.

In BLAISE, there are two ways for Kernels to be invoked. First, the root of the Kernel hierarchy can be applied to the root of the State hierarchy, which results in the State hierarchy advancing one step in the Markov Chain. Alternately, Initialization Kernels may be invoked in response to pieces of State being created or destroyed. Whenever a new piece of State that may need to be initialized (such as a new component in a mixture model) is created and attached to the State hierarchy,

the Virtual Machine determines where in the State hierarchy the new State was attached and looks up whether any Initialization Kernels were configured to handle initializations matching this location. If so, the Initialization Kernel is invoked on the new State. Likewise, when that State is later removed from the State hierarchy, the Virtual Machine will look up the same Initialization Kernel and once again invoke it on the new State, this time passing the Initialization Kernel a flag indicating that it should De-initialize this State.

5.7 Discussion

In this chapter I supported my thesis by implementing a virtual machine for the BLAISE SDK modeling language. This virtual machine is implemented in Java and runs on a wide variety of off-the-shelf hardware, making BLAISE an accessible framework. In this chapter I also identified several issues that had to be resolved in order to ensure that the BLAISE VM could execute even large SDK models efficiently. Handling these improperly could cause the execution time to grow inappropriately with the size of model. For example, without Density memoization, the execution time for evaluating the root Density would grow with every Density added to the graph; with Density memoization, execution time only reflects the changed densities. This is typically orders of magnitude faster; for example, if only one Density is attached to a State that changes value, then only that Density and its ancestors in the Density tree need to be reevaluated. Similarly, without incremental updates, the execution time of a Multiplicative Collection Density grows with the number of children of that Density; with the appropriate incremental update scheme, the execution time is independent of the number of children.

For each of the efficiency issues I identified, I described how I solved these issues in BLAISE and provided benchmarks to quantify the improvements gained from these optimizations. In each case, the optimizations are transparent to the end user, who only needs to think about the BLAISE SDK formalism and can allow the virtual machine to handle the efficient implementation. For example, transactions were used

to make Metropolis-Hastings and Gibbs Sampling significantly more efficient, but the end user never needs to worry about these issues because the transactions are handled automatically by the implementations of the M-H and Gibbs Kernels supplied by the BLAISE VM.

The algorithms and data structures described here yield significant performance improvements and enable the ease-of-modeling provided by the SDK formalism. However, implementing these features requires the entire system to be built with them in mind. As a result, one-off implementations – that is, starting from scratch every time a model and inference algorithm are to be implemented, a standard practice in the field – almost never include such features. BLAISE brings these efficient algorithms and data structures to every model built in BLAISE, so even simple models run faster, and can be grown into more sophisticated models without being rewritten.

Chapter 6

Applications in BLAISE

My thesis is that a framework for probabilistic inference can be designed that enables efficient composition of both models and inference procedures, that is suited to the representational needs of emerging classes of probabilistic models, and that supports recent advances in inference.

In this chapter, I support this thesis by describing several applications that have been built using the BLAISE framework. The applications presented in this chapter fall into three categories. Some of the applications are domain-specific models of moderate sophistication that benefit largely from BLAISE’s rapid development, extensibility, and inference-enhancing transformations. These applications include a generative model for computer vision, and a model for analyzing neurophysiological data. Other applications in this chapter highlight BLAISE’s support for very sophisticated models, in which there are multiple unknown-sized sets of objects interacting in interesting ways. These applications, including models for relational data and for topic analysis, benefit from BLAISE’s complexity-localizing abstractions and exercise all aspects of the SDK formalism. Finally, this chapter presents three examples of other modeling languages built on the BLAISE framework: standard graphical models (e.g. Bayes nets, Markov Random Fields, factor graphs), a reimplementa-tion of the BUGS language [58, 62], and Church [20], a stochastic extension of Scheme. Embedding these modeling languages in BLAISE demonstrates the wide coverage of BLAISE

models.

Because BLAISE is intended to be used as a tool by the probabilistic inference community, it was important to evaluate whether BLAISE is useful to researchers other than myself; therefore, several of the applications described in this chapter are the result of collaborations with colleagues. In these collaborations, I provided BLAISE training and support, as well as the implementation of all general aspects of BLAISE described in chapters 3 through 5, while my collaborators provided domain expertise. I explicitly delineate my role in each of these applications.

6.1 Generative Vision

Author's role: Modeler, Implementation Lead

One of the first applications developed with BLAISE was a simple exploration into generative models of vision. Hermann von Helmholtz initiated scientific inquiry into visual perception in the nineteenth century with the idea that vision constitutes inference from incomplete data to the most likely explanation of that data. However, only recently have probabilistic models of the human visual system been developed that fully embrace this idea [67]. In part, this is because even extremely simplified models of vision tend to be sophisticated in their modeling and inference requirements.

This application explores a generative model of vision in a toy context as a proof of concept that BLAISE could support work in this area. The goal of this application is to identify and locate simple solids in a moderate resolution (320x240 pixel) rendering of a 3-dimensional scene. The model assumes a simple generative explanation of how images are formed: first, a number of simple solids are generated in 3-space. This scene is then rendered to an image using standard computer graphics. Finally, each pixel is independently corrupted by a noise source to produce the final image. Visual inference in this model is then a matter of inferring from the final image the best explanation (or a distribution over good explanations) for how many of what kinds of simple solids are in the scene and where they are located.

This generative description of the model translates directly to a BLAISE SDK graph. I restrict the model to two kinds of simple solids: spheres (of fixed radius) and blocks (with fixed dimensions, but of unknown axis-aligned orientation). Each solid is represented as a Composite State containing real-valued States X , Y , and Z , representing the location of the solid in 3-space; the Composite State for blocks also contains a discrete (6-valued) State *Orientation* representing the orientation of the block. The scene is then represented as a Composite State containing a Collection State of spheres and a Collection State of Blocks. The data (i.e. the input to vision) is implemented as a Compound State containing an image State as well as real-valued States X , Y , Z representing the location of the camera (and assuming that the camera is looking in the positive z direction, with other camera parameters known *a priori*). The State hierarchy is completed using a Compound State containing the scene State and data State just described.

The Density hierarchy for the model has a similar structure to the State hierarchy. For each solid, there is one Density connected to each of X , Y , and Z representing the prior on location; for this simple model, uniform Densities were used, representing equal probability over a rectangular prism of 3-space. For blocks, there is also a Density connected to the block’s *Orientation* state which assigns equal probability mass to each possible orientation. For each solid, all of the priors are composed using a Multiplicative Density D_{solid} . Next, each Collection State (i.e. the “spheres collection” and the “blocks collection”) has a parallel Associated Collection Density that manages the D_{solid} Densities for that object type. Each Collection State also has a Density representing the prior on the number of objects of that type; for this implementation, a geometric distribution with known parameters was used. The two Densities associated with each Collection State are composed using a Multiplicative Density $D_{\{spheres\}}$ or $D_{\{blocks\}}$. The Density on the Scene is completed using a Multiplicative Density D_{scene} containing $D_{\{spheres\}}$ and $D_{\{blocks\}}$. Finally, a single Density $D_{image|scene}$ connects the scene State to the image State, and a Multiplicative Density containing D_{scene} and $D_{image|scene}$ is used as the root Density. $D_{image|scene}$ computes the probability $p(image|scene)$ by rendering the scene from the camera’s viewpoint,

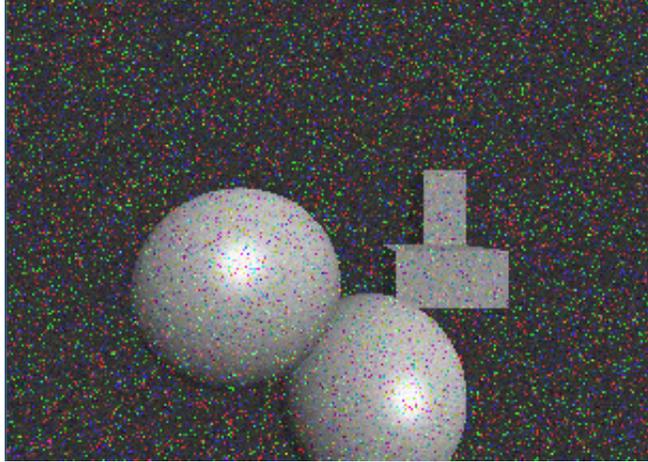


Figure 6-1: Input for the toy generative vision exploration described in section 6.1. The inference goal is to explain the image using appropriately located and oriented blocks and spheres. This input was generated by forward sampling from the model prior.

using OpenGL and off-the-shelf graphics hardware, then computes the probability that each pixel in the target image was generated by adding pixel-wise Gaussian noise to the rendered image.

For inference, I use simple stock-BLAISE Kernels to adjust the parameters of the simple solids, as well as birth/death Kernels to create and destroy solids. To adjust real-valued parameters (i.e. X , Y , Z), I use Metropolis-Hastings Kernels with Gaussian Perturbation Kernels; that is, if the current value of the State is x , the proposal will be sampled from $Normal(\mu = x, \sigma = \sigma_0)$. For discrete parameters (i.e. *Orientation*), enumerative Gibbs Kernels will be used. The birth/death Kernels create (or destroy) the appropriate States for new solids, using initialization Kernels to sample values for the parameters from the uniform prior. It is worth re-emphasizing that these inference Kernels are completely isolated from the implementation of the Density, with its OpenGL complexity. For example, the Kernels that perform inference on the object locations are the the same Kernels that were used to infer mixture model component parameters in section 3.4.2.

The execution time of this model is dominated by the OpenGL rendering step. The BLAISE Virtual Machine's transaction and caching systems are crucial for maintaining efficiency. This model has a large number of non-optimal local optima. For

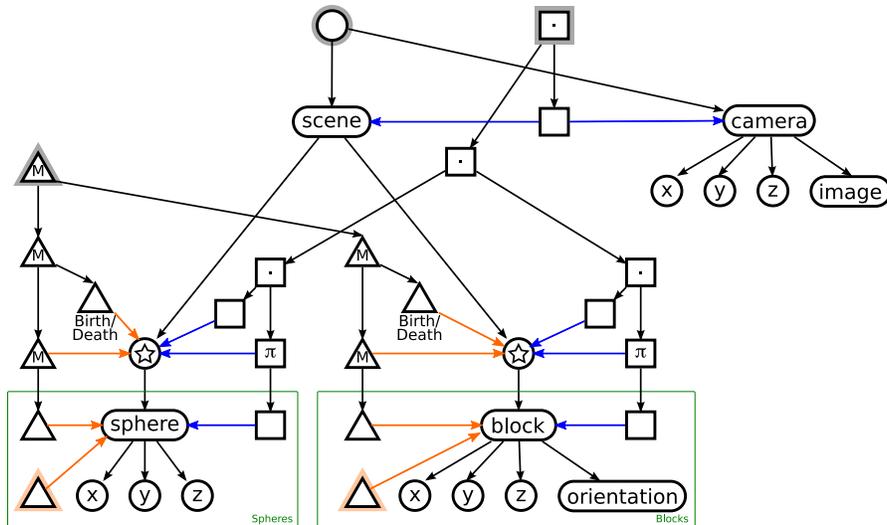
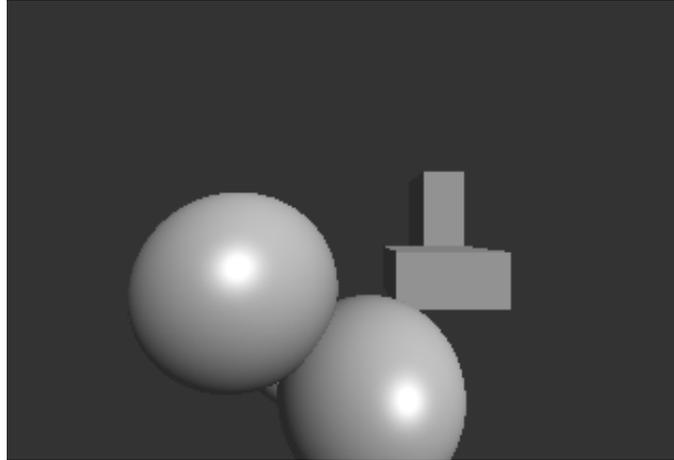


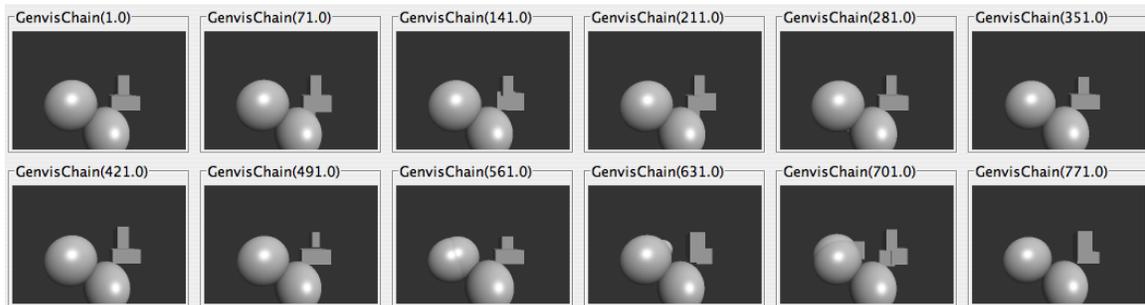
Figure 6-2: The BLAISE SDK diagram for the toy generative vision model described in section 6.1. For legibility, the Kernels for inference on the parameters of the spheres and blocks are abstracted away in this diagram.

example, it is easy for two blocks positioned side by side to render very similarly to a single block. This model therefore provides challenges for implementing an inference kernel that mixes well. Fortunately, BLAISE has an automatic tool for handling such situations: the parallel tempering transformation. Parallel tempering the model makes it significantly easier to escape local optima and greatly reduces the number of samples required.

A number of variations to this model are also possible. For example, stereo (or even multi-camera) vision is simply a matter of adding more data States (the compound state containing the image State and the eye location) as well as the appropriate $D_{image|scene}$; the inference Kernels stay exactly the same. Localization would also be possible using this model by using the the same State and Density hierarchies, but instead of pre-setting the camera location, the scene is completely pre-set. The Kernel hierarchy would then be rewritten to adjust only the camera location; these Kernels are just like the Kernels used to adjust the X , Y , and Z of a solid.



(a) Cold chain



(b) All chains

Figure 6-3: The Generative Vision model is prone to local optima; for example, two adjacent blocks render very similarly to a single block closer to the camera. Parallel tempering is required for MCMC on this model to mix well. (a) shows the current state in of the cold chain after 5000 samples when the model has nearly converged on ground truth. (b) shows the current state in each of 12 parallel tempered chains after 5000 samples; each chain is labeled with its temperature.

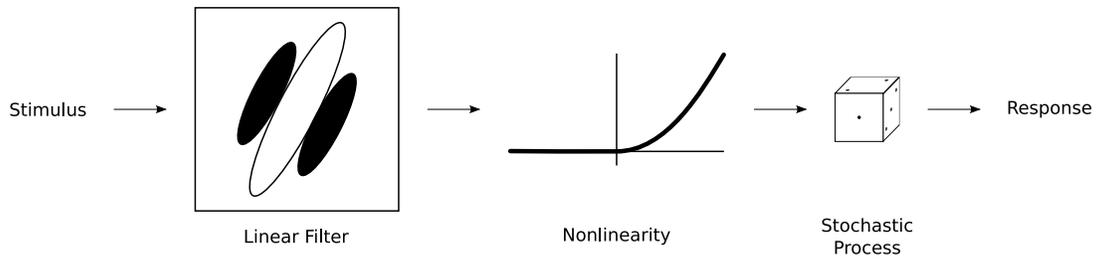


Figure 6-4: A common functional model for a neuron in the primary visual cortex (V1). The information input for the neuron (the stimulus) is a time-varying image. The stimulus is convolved with a linear filter, such as a Gabor filter. This signal then undergoes a nonlinear transform, and finally a stochastic process, such as a Poisson process, generates a neural spike train response that can be detected experimentally. (Adapted from [55])

6.2 Analysis of Neurophysiological Data

Author's role: Support

Neurophysiologists seek to understand how the brain embodies the mind by first understanding how individual neurons in various regions of the brain respond to stimuli. For example, a neurophysiologist studying the V1 region of the visual cortex (a region specialized for extracting basic information about visual scenes) might be interested in characterizing the neural response to spatiotemporal patterns in visual input (e.g. [55]). A simple functional model of such neurons, the linear-nonlinear-Poisson (LNP) model, is shown in figure 6-4. More complicated models, such as the generalized LNP model [55], include multiple linear filters, each with a distinct nonlinearity transform, as well as a nonlinear combination function to combine the signals from each filter before generating the spike train.

Analyzing experimental data requires inferring model parameters governing the linear filter (for example, spatial orientation and wavelength of a Gabor function) and the shape of the non-linear function. Traditional approaches to this analysis have used frequentist statistical analyses of the mean and covariance of that portion of the stimulus which occurred in a temporal window immediately preceding a spike to attempt to fit the model to observer responses. This “spike-triggered” analysis is complicated due to the non-linearities in the model [55]. Only in the last year have

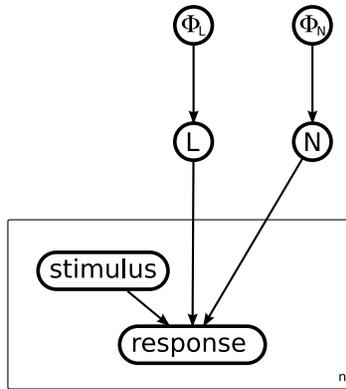


Figure 6-5: The LNP functional model for a V1 neuron, cast as a generative model in Bayes net notation. The L variable represents the linear filter parameters, governed by hyperparameter Φ_L . The N variable represents the nonlinearity parameters, governed by hyperparameter Φ_N . Each instance of the plate captures an experimental trial, with stimulus variables capturing the stimulus that was presented during that trial and response variables capturing the neural response. The conditional probability distribution $p(response|stimulus, L, N)$ captures the probability that the stochastic process (i.e. the Poisson process) would produce the given response when presented with the stimulus.

Cronin et al. [12] developed general tools to perform a principled Bayesian analysis of these experiments (but see [56]). These tools treat the neural functional model (e.g. LNP or generalized LNP) as a Bayesian generative model and use BLAISE to estimate the conditional distribution on the parameter values. These tools have a number of advantages over existing techniques:

- Bayesian priors allow the incorporation of other prior knowledge (for example, soft constraints on the shape of the nonlinearity).
- Bayesian model selection supports inferences about the functional form the linear filter or nonlinearity.
- Probabilistic inference produces a distribution on parameter values, allowing easy assessment of confidence in a particular parameter value
- Unlike the traditional frequentist statistical analyses, a wide range of models can be tested using the same techniques. For example, inference for the generalized LNP model is very similar to inference for the LNP model. The same framework can also support inference in models which have not yet been considered in the literature, due to the difficulty of performing parameter inference.

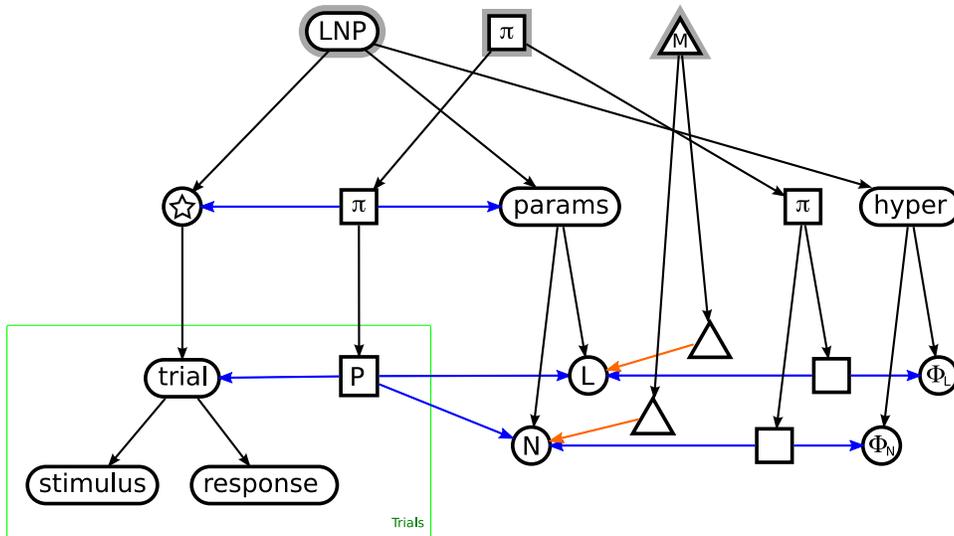


Figure 6-6: The same generative model from figure 6-5, case as a BLAISE model, including inference Kernels for the linear filter and nonlinearity parameters. This model is a slightly idealized version of the BLAISE model actually used in [12, 57, 41]. For example, the actual model included additional support for media descriptors to describe the type of media in the stimuli and support for the same stimulus instance to be shared across multiple trials in order to minimize storage requirements.

Implementing Bayesian methods for LNP requires the use of modeling components that typical modeling packages do not include. For example, evaluating the model’s Density requires convolving a linear filter with the stimulus and computing the non-linear transform of this convolution. BLAISE SDK modeling language supports these requirements through the abstraction barrier between Densities and Kernels; all the standard inference techniques that ship with BLAISE continue to work, even with custom Densities. The BLAISE VM makes it easy to implement custom Densities, and the transactional caching in the VM automatically minimizes the number of potentially expensive custom Density evaluations. The flexible infrastructure has also allowed Cronin et al. to provide a Matlab interface to these tools, allowing a model to be fit with an instruction as simple as:

```
tc_sample(x, y, 'circular_gaussian_360', 'poisson')
```

where x is a vector of stimuli, y is a vector of responses, `circular_gaussian_360` determines the functional form of the linear filter and non-linearity, and `poisson` selects the stochastic process.

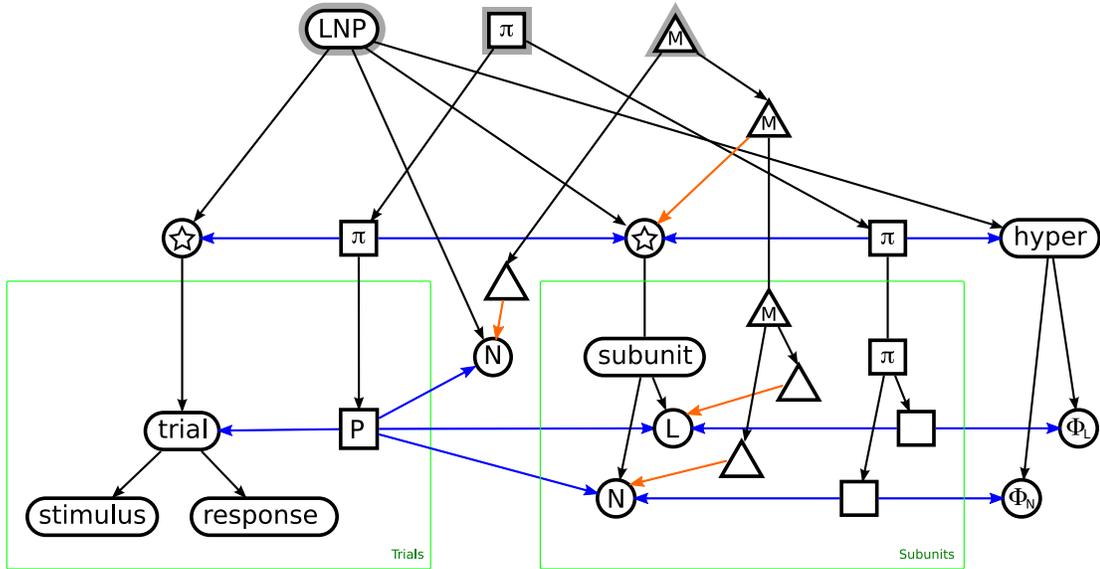


Figure 6-7: An extension of the model LNP model from figure 6-6, this model implements inference for the generalized LNP functional model [55] with a fixed number of linear filter subunits. Generalized LNP was also implemented by Cronin et al. in BLAISE. This model could be extended to infer the number of subunits by adding a prior on the number of subunits and including a subunit birth/death kernel.

The BLAISE-based tools developed by Cronin et al. are already advancing the field of neuroscience. For example, Schummers et al. [57] used these tools to examine the temporal dynamics of orientation tuning of V1 neurons in cats, finding that the tuning parameters change over the time-course of a neural response in more than forty percent of V1 cells, and that these temporal dynamics are influenced by the cell’s location within the cortical network.

A major challenge in systems neuroscience is to determine how neuronal classes identified on the basis of morphology or gene expression patterns perform different functional roles; the Blaise-based toolbox is ideal for testing hypotheses about the nature of these functional differences. Mao et al. [41] used these tools to investigate the roles of different types of inhibitory interneurons in the production of stimulus-specific responses (e.g. orientation-selective responses) in the visual cortex, finding that reducing the number of caltrenin-positive neurons in the visual cortex of mice resulted in fewer orientation-tuned V1 cells.

In response to developing these analysis tools using BLAISE, Cronin reported the

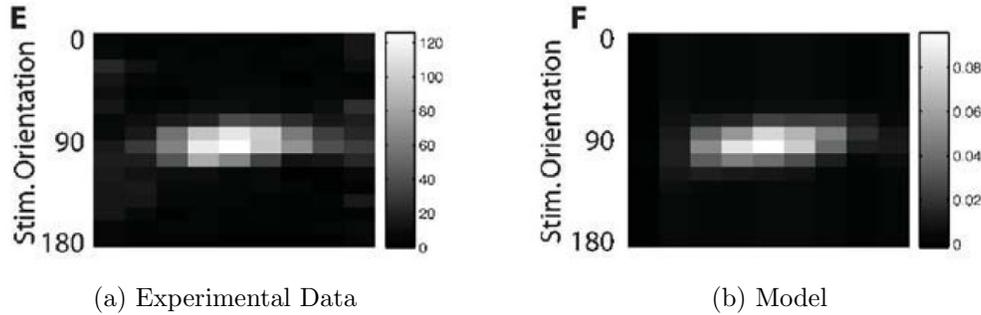


Figure 6-8: Schummers et al. [57] used BLAISE-based tools to model the temporal dynamics of orientation tuning of V1 neurons in cats. (a) The recorded time-varying response of a neuron to different stimulus orientations. The horizontal axis is time from the presentation of the stimulus. The vertical axis is the stimulus orientation. The plotted data is the number of recorded cell spikes (binned in 5ms intervals). Note that high-spike-count region of the graph is slightly angled, indicating that the cell’s orientation changes over time. (b) The model estimated using BLAISE-based tools. The plotted data is the firing rate predictions from the model. Reproduced from [57] (figure 4E-F) with permission.

following:

Before Blaise, I had implemented (and reimplemented) a number of samplers for my data analysis models. At a certain point, I realized that there was a rather large class of models that I would like to be able to represent and upon which I would like to perform inference. While I understood the domain well, the prospect of creating a general, flexible, and robust implementation of the entire class of models was daunting.

It was at this point that I was introduced to Blaise, which (even in an early version) offered an extremely powerful set of modeling and inference tools. The Blaise abstractions and library allowed me to quickly develop a general implementation of my model class, and the resulting software has held up very well to a number of practical tests. Most importantly, a number of other members of my lab now use this software toolbox, BayesPhys, to analyze their data, and a number of forthcoming publications will use it when answering crucial questions about neurophysiological data. It would simply not have been possible to accomplish these goals without Blaise - not even a grad student has that much time.

~ Beau Cronin

6.3 Relational Models

People are remarkably adept at making appropriate generalizations – extracting patterns from their experiences, and using these patterns to make predictions about novel

scenarios. If cognitive scientists are to understand human cognition, they must understand the mechanism that enable such generalizations. Similar challenges emerge in any number of industrial settings, where the ability to gather and store data has outstripped our ability to process the data automatically. Even with human assistance, there is simply too much data to manage. Further complicating matters is the fact that in many industrial databases are very sparse and noisy: most of the data that could theoretically be gathered is missing (perhaps impatient users are only willing to provide a few pieces of information each), and what data is present is often gathered through some noisy medium (such as subjective user ratings). If such industries are to make effective use of their data, they need ways to generalize so they may fill in the missing elements, and they need ways to extract concise patterns from the data, so that humans may work with the hidden structure underlying the data rather than the overwhelming raw data itself.

For interpreting the properties of a single type of object, mixture models are often used. Mixture models partition the individual objects into groups whose features can be explained in the same way. These groups form the underlying structure of the data and license inferences about unobserved properties (because same-group objects share a feature model).

Probabilistic relational models are used to model relationships between entities (objects); that is, the kind of data that might be found in a relational database. Relational models are defined over domains of entities; for example, a relational model for a movie rental company might have two domains: *Users* and *Movies*, where the entities in the *Users* domain are the various users (e.g. Alyssa, Ben, Louis, etc.) and the entities in the *Movies* domain are various movies (e.g. Monty Python and the Holy Grail, The Life of Brian, Real Genius, etc). Each relation model also has a number of relations, defined as a mapping from an ordered tuple of domains to a type of data. For example, the movie rental company might have a relation for movie ratings, defined as

$$Ratings \triangleq Users \times Movies \rightarrow \{0, 1, 2, 3, 4, 5\}$$



indicating a mapping of $\langle User, Movie \rangle$ pairs to the rating that user gave that movie, represented as an integer between 0 and 5. If the movie rental company also operated a social networking site, the relational model might also have a relation

$$SocialNetwork \triangleq Users \times Users \rightarrow \{\text{Likes, Dislikes}\}$$

indicating a mapping of $\langle User, User \rangle$ pairs to whether or not the first user likes the second user. Given a set of domains and relations defined on those domains, a relational model is a probabilistic model of how specific data was generated. Continuing the movie example, a relational model might try to explain data such as:

<i>Ratings</i>	<i>SocialNetwork</i>
$\langle \text{Alyssa, Monty Python and the Holy Grail} \rangle \rightarrow 5$	$\langle \text{Alyssa, Ben} \rangle \rightarrow \text{Like}$
$\langle \text{Alyssa, The Life of Brian} \rangle \rightarrow 4$	$\langle \text{Ben, Alyssa} \rangle \rightarrow \text{Like}$
$\langle \text{Ben, Monty Python and the Holy Grail} \rangle \rightarrow 4$	$\langle \text{Ben, Louis} \rangle \rightarrow \text{Dislike}$
$\langle \text{Ben, Real Genius} \rangle \rightarrow 4$	$\langle \text{Alyssa, Louis} \rangle \rightarrow \text{Dislike}$
$\langle \text{Louis, Monty Python and the Holy Grail} \rangle \rightarrow 1$	$\langle \text{Louis, Alyssa} \rangle \rightarrow \text{Like}$
\vdots	\vdots

Relational models typically operate by assuming additional structure among the entities; for example, a model might assume that each domain can be partitioned into groups of entities that behave similarly. This structure is an unobserved variable that must be inferred, but it is this structure that licenses generalization for inferences such as: “Assume a new user Lem joins the system, and the only information you have on Lem is that Alyssa dislikes him. Would Lem give Monty Python and the Holy Grail a high rating, if he were to rate it?”

Recently developed relational models are quite sophisticated. They involve inference over structured representations including partitions and trees, they use non-parametrics (for example, to allow the number of entity groups to grow as justified by the data), and they benefit from sophisticated inference techniques to explore the model space efficiently. In this section, I present two relational models that have been implemented in BLAISE: the Infinite Relational Model and the Annotate Hierarchies

relational model.

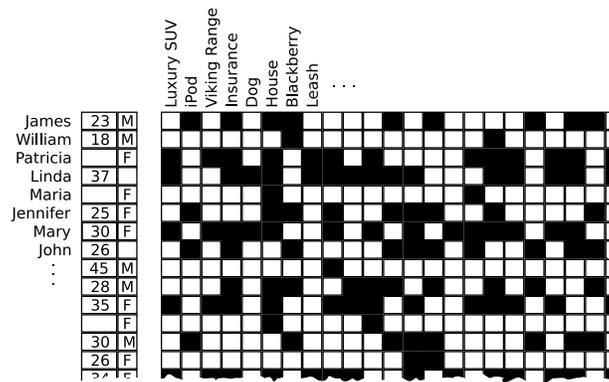
6.3.1 Infinite Relational Model

Author's role: Implementation Lead

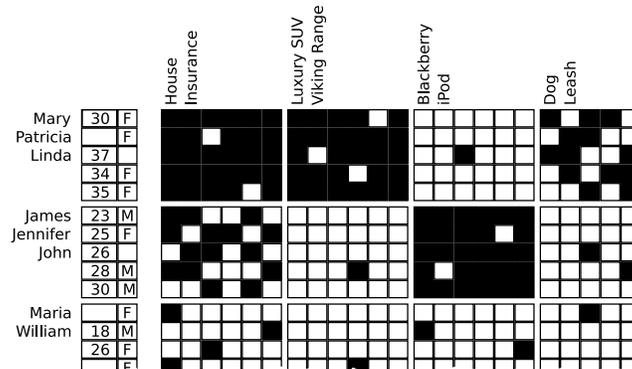
The Infinite Relational Model (IRM) [30] assumes that each domain of entities is partitioned using a Chinese Restaurant Process (see figure 6-9). Viewing each relation as a matrix, the domain partitions¹ divide the matrix into blocks, with the blocks playing the same role as components in a mixture model. For example, if the *Movies* relation were being modeled using a Beta-Binomial model, each block would have an independent Beta-Binomial model to explain the datapoints assigned to that block, just as each component of a Beta-Binomial mixture model would have an independent Beta-Binomial model to explain the datapoints assigned to that mixture component. The BLAISE implementation of the IRM draws on this observation that an IRM's relation model is, in essence, a multi-dimensional extension of a mixture model (see figures 6-10 through 6-14).

The state space for the Infinite Relational Model is deceptively large. There are B_n distinct ways to partition a domain of n objects into disjoint non-empty subsets, where B_n is the n^{th} Bell number [53]. The Bell numbers grow quite quickly; for example, $B_{10} = 115975$. In the IRM, there are multiple domains, so the size of the state space (putting aside any variables representing parameters associated with each mixture component) is the product of the Bell numbers for each domain. The examples in figures 6-12 and 6-13 each have two domains of 300 objects; thus the size of the state space is $(B_{300})^2$, or approximately 10^{908} . It is therefore striking that in just 100 inference sweeps, comprising $6 \cdot 10^5$ entity visits, MCMC inference can converge as in figures 6-12b and 6-12c.

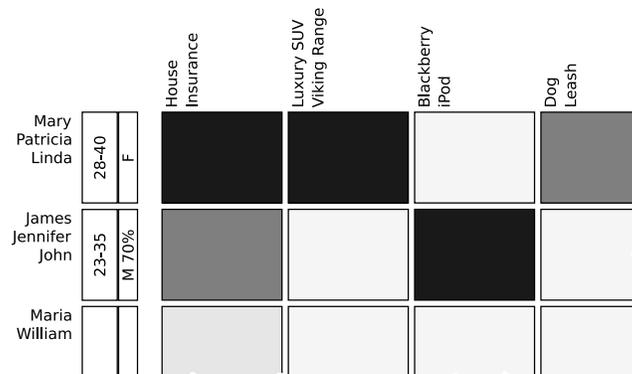
¹To be clear, each domain is partitioned exactly once, even if the domain is used in multiple relations or if the domain is used multiple times in the same relation.



(a) Input



(b) Partitions



(c) Component Models

Figure 6-9: Relational models take as input relational data, such as the mock customer purchase data in (a). The Infinite Relational Model explains this data by reordering and partitioning the entities in each domain (b) and generating a mixture model component for each partition-induced block of data (c).

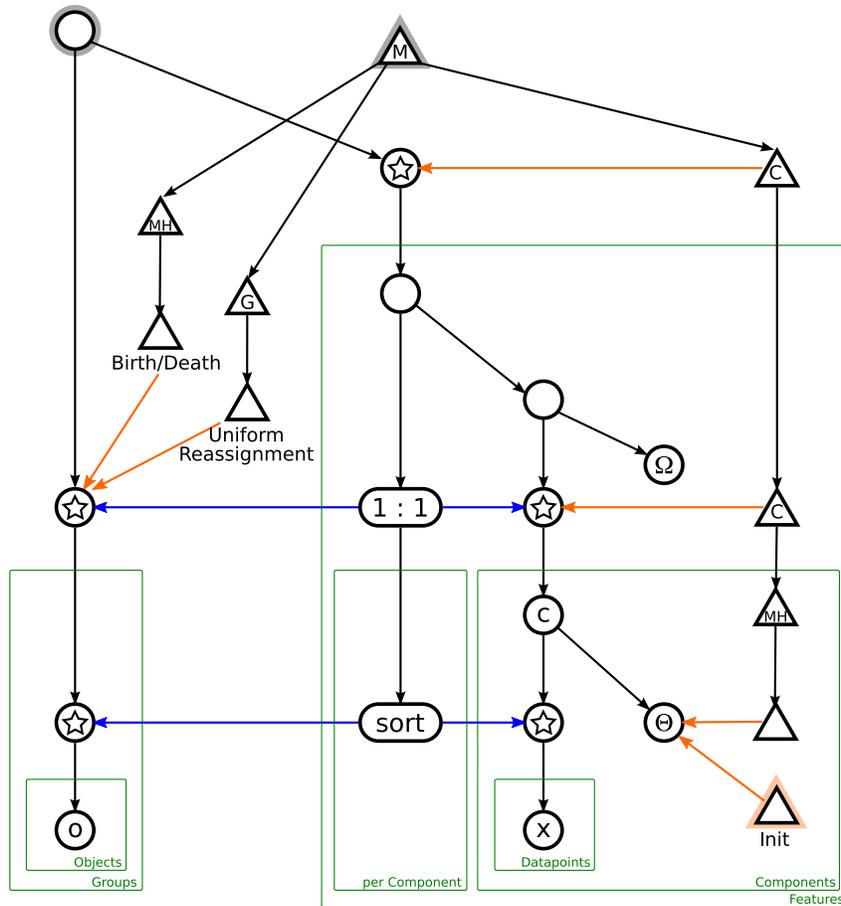


Figure 6-10: The multi-feature mixture model presented in chapter 3, reproduced from figure 3-26. The IRM model will be an extension of this model.

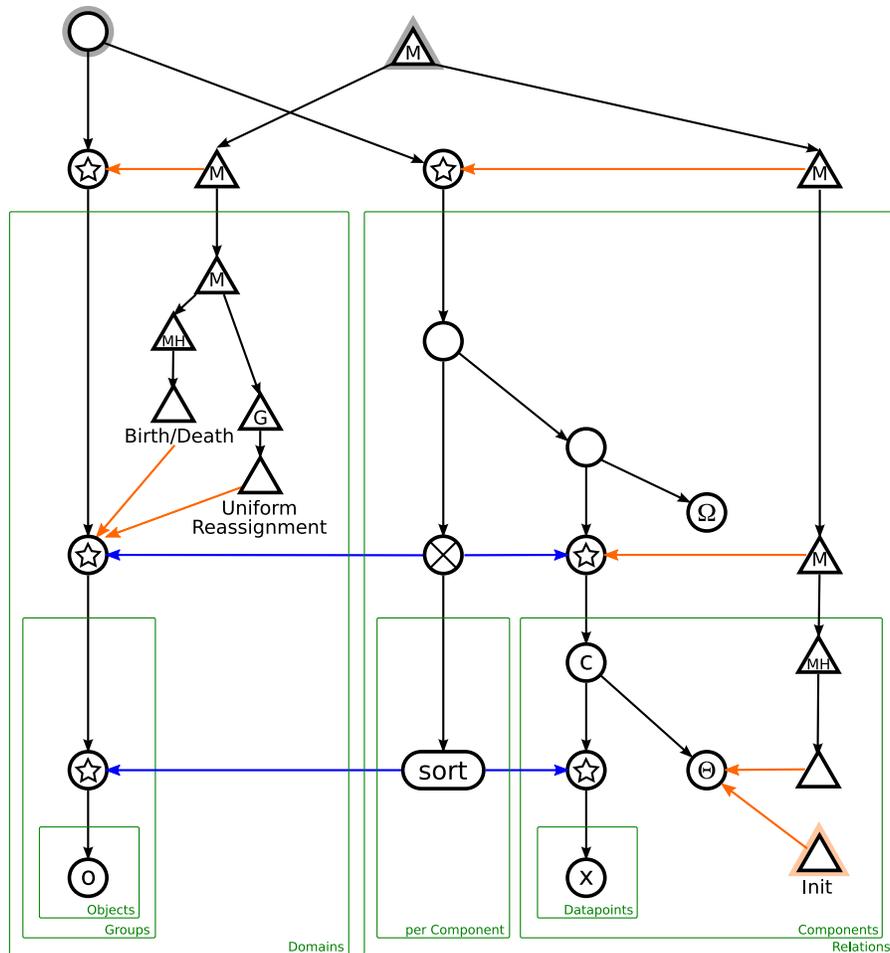


Figure 6-11: The Infinite Relational Model (IRM) State and Kernel structures. The primary difference between the IRM and the multi-feature mixture model presented in figure 6-10 is that there are multiple domains of objects on the left side of the model, with a Collection State containing all the domains. The 1 : 1 Constraint State from the mixture model has also been replaced with a Cartesian Product state. The Cartesian Product state is a standard BLAISE State that has edges to multiple object partitions, and ensures that for each tuple of object groups drawn from those partitions, there is a corresponding component. For example, a *Users* \times *Movies* relation in an IRM would have a Cartesian Product State with one edge to the *Users* domain and one to the *Movies* domain; this State would ensure that every $\langle UserGroup, MovieGroup \rangle$ pair had a corresponding component associated with it.

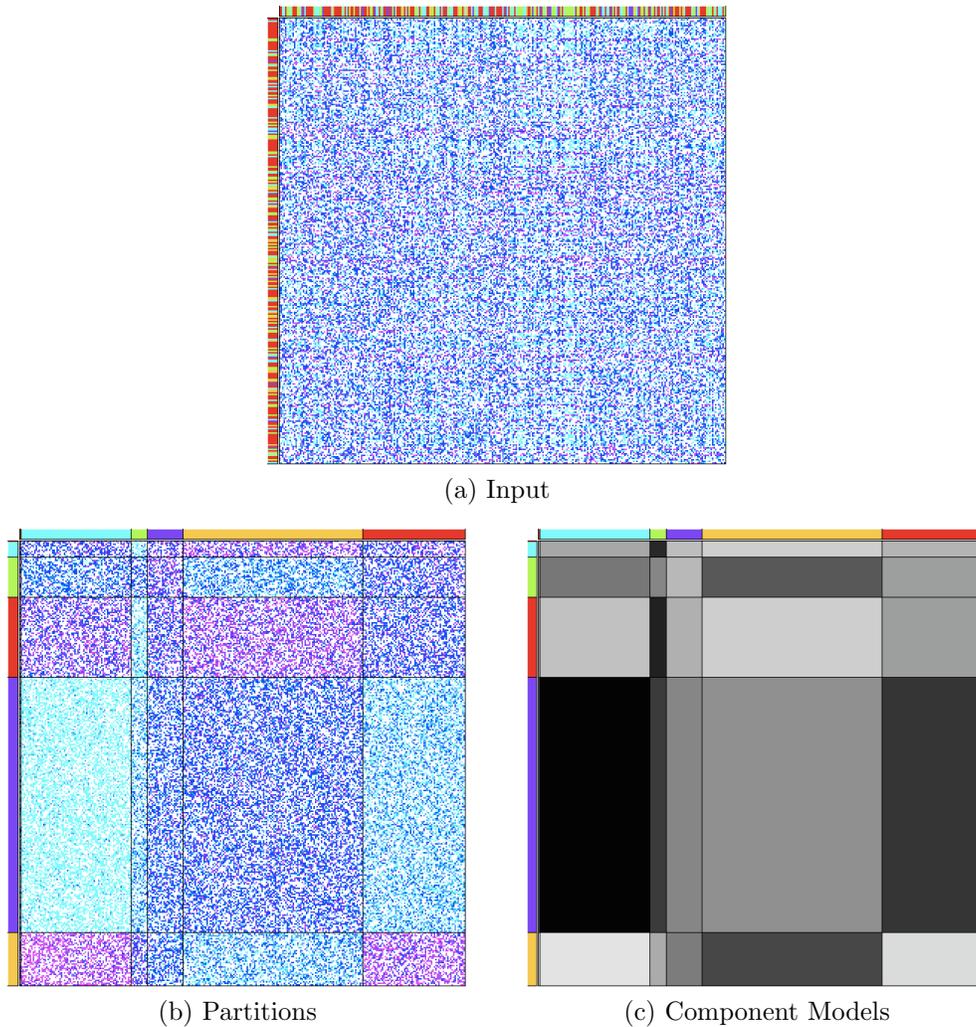
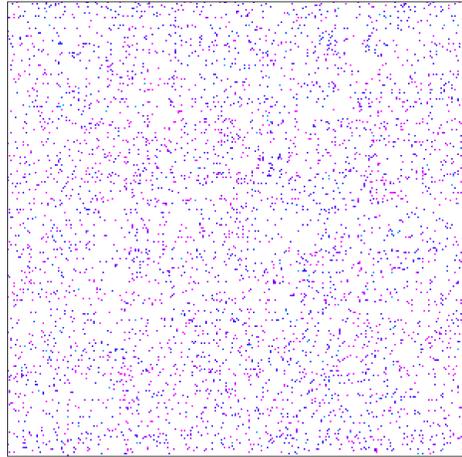
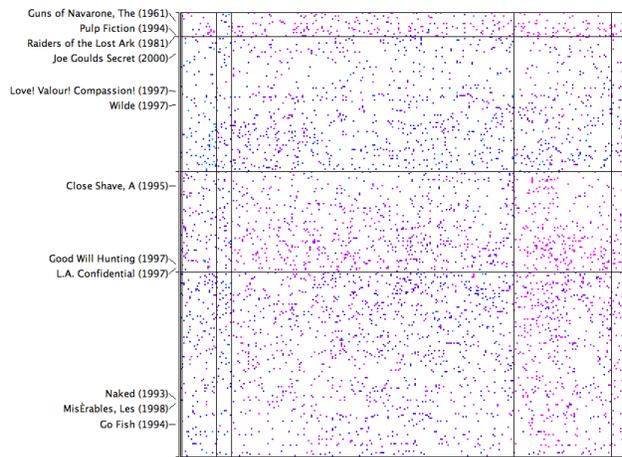


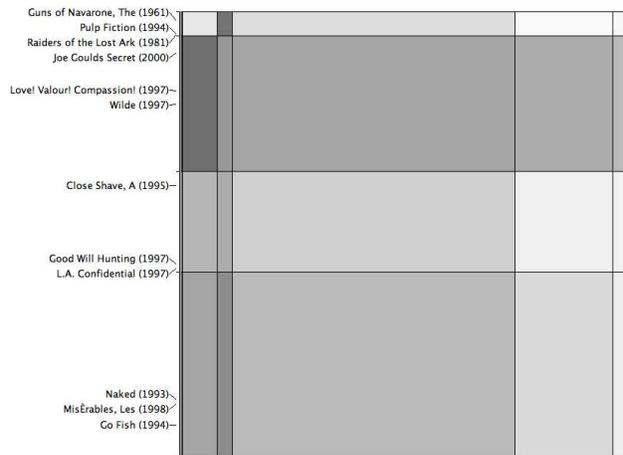
Figure 6-12: In this figure, the Infinite Relational Model is applied to a randomly generated synthetic dataset, where each datapoint is an integer on the range 0–5. This dataset is meant to be an easy-to-interpret version of the Movielens [25] dataset used in figure 6-13. (a) shows the 300×300 entity synthetic input. To generate the input, each domain was partitioned into 5 entity groups. Parameters for each IRM component’s beta-binomial model were sampled from a prior distribution, then 50% of the datapoints were observed by sampling values from the component model. Observed datapoints are colored in a teal–blue–magenta palette, where teal=0 and magenta=5; missing data is left white. The entity order was then randomized before being presented to the algorithm; the ground truth partition assignments are color coded in the ruler bars to the top and left of the data matrix (these values were not available to the algorithm). After a 100-sweep run of my BLAISE-based IRM implementation, the partitions in (b) were found. The mean value of the component models are seen in (c), where black=0 and white=5. The ground truth was recovered almost perfectly, as seen in the ruler bars.



(a) Input



(b) Partitions



(c) Component Models

Figure 6-13: The Movielens [25] dataset is a real-world dataset with a $Users \times Movies$ relation, where each datapoint is a user's rating of a movie on a 0-5 scale. In (a) shows a $300 \text{ user} \times 300 \text{ movie}$ subset of Movielens, where movies are on the vertical axis and movies are on the horizontal axis. Only about 4% of the possible datapoints are present in the Movielens dataset. (b) and (c) show the result of a 200 sweep inference run.

Infinite Relational Model Runtime Analysis

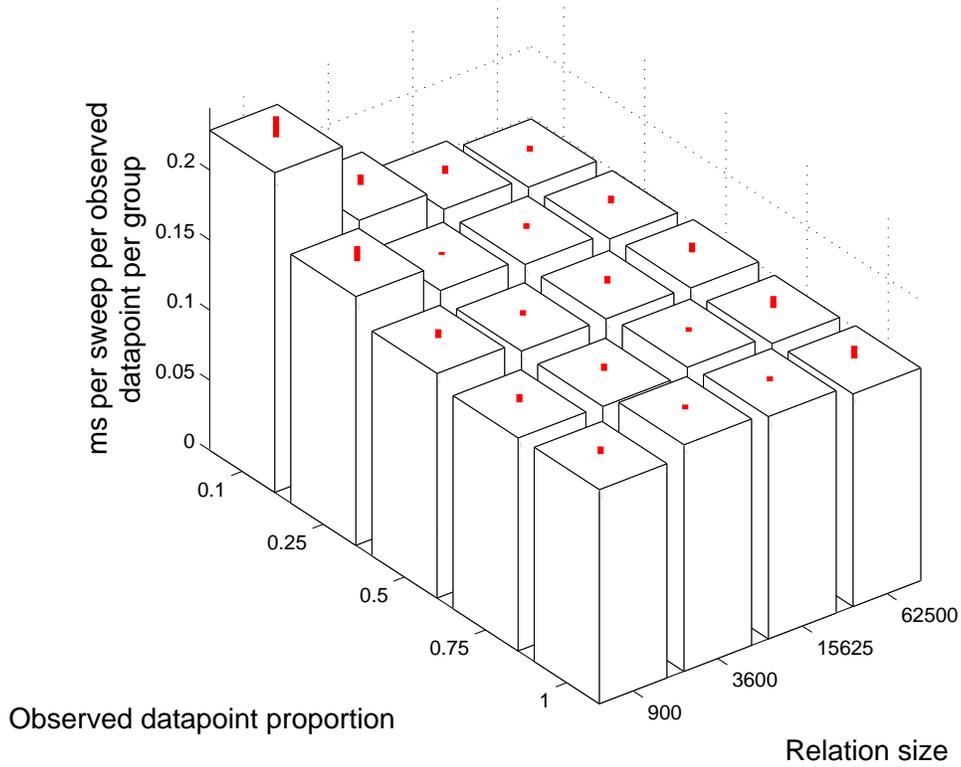


Figure 6-14: Recent analysis [39] suggests that proper data structures and caching should allow Gibbs-sampling-based inference in an Infinite Relational Model to be performed in $O(g \cdot dp \cdot arity^2)$ time, where g is the number of groups in each domain, dp is the number of observed datapoints, and $arity$ is the relation arity. This figure shows timing results from a BLAISE-based IRM, using a beta-binomial conjugate model as the component model for a single relation over two domains. The model was run for 200 sweeps of inference over all the entities. The data collected was the average number of groups over the course of inference (\tilde{g}) and the total runtime t . Data was collected for a variety of dataset sizes; in the plot, relation size is the number of datapoints that could be observed in the relation (i.e. the product of the two equal-sized domains), while observed datapoint proportion is the fraction of those datapoints that were actually observed (thus dp in the timing estimate is $size \cdot proportion$). If this implementation is $O(g \cdot dp \cdot arity^2)$, it should be the case that (for a specific arity) $t/200 = t_{dp} \cdot \tilde{g} \cdot dp$, where t_{dp} is constant for all runs and represents the time per sweep per observed datapoint per group. In the plot, bars indicate mean value of t_{dp} over 10 runs; whiskers indicate standard deviation. Each run used a different randomly generated synthetic dataset with 5 groups in each domain. Observed \tilde{g} values ranged from approximately 5 for the smallest models to 10 for the largest models, indicating that larger models require more sweeps to converge. Results show that t_{dp} is constant, indicating that the BLAISE implementation of the IRM is $O(g \cdot dp \cdot arity^2)$, at least over several orders of magnitude for dp . Note that the smallest models show a slightly non-constant t_{dp} ; this is most likely the result of Java's Just-In-Time compilation.

6.3.2 Annotated Hierarchies

Author's role: Support

The Infinite Relational Model makes several potentially over-restrictive assumptions about the structure of the data being modeled. First, it assumes that the entities in each domain are divided into flat partitions – that is, that every entity belongs to exactly one group. When a domain participates in multiple relations, the IRM also assumes that the domain is partitioned in the same way for each domain it participates in. Finally, the IRM assumes that domain partitions carve the relational data into a regular grid – there is no possibility that a *Users* \times *Movies* relation would have one group of users for which there are three relevant movies groups, while another group of users has only two relevant movie groups.

The Annotated Hierarchies model for relational data [54] relaxes all these assumptions. Instead of assuming a flat partition on each domain, the Annotated Hierarchies model assumes that entities in each domain are assembled into trees, where the leaves of the tree are the entities and the inner nodes are nested subgroups of entities. Entities are then partitioned into a flat group structure by selecting a set of inner nodes from the tree, such that the nodes cover all the entities without overlap; this is called a tree-consistent partition. When a domain participates in multiple relations, the same tree is used each time, but a different tree-consistent partition is used; that is, the domain may be partitioned differently for each use in a relation, but all the domain partitions will be consistent with the single tree for that domain. The Annotated Hierarchies model also relaxes the assumption that the relational data must be carved into a regular grid. In the *Users* \times *Movies* example, it would be possible for one group of users to have one tree-consistent partition on movies, while a different group of users used a different tree-consistent partition. In fact, every time relational data is subdivided, the resulting subdivisions are treated as independent with respect to further partitioning.

Annotated Hierarchies is a very sophisticated probabilistic model, and inference in this model is likewise challenging. Roy et al. [54] implemented Annotated Hierarchies

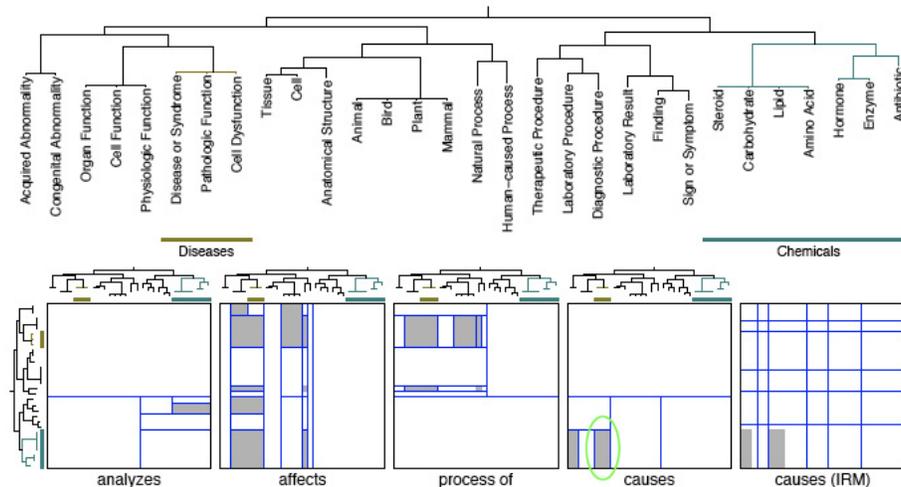


Figure 6-15: Roy et al. [54] used BLAISE-based tools to fit the Annotated Hierarchies model to relational data from the Unified Medical Language System (UMLS) data set [42]. In this dataset there was only one domain D of entities, with 49 binary relations on these entities. For example, the binary relation $Causes \triangleq D \times D \rightarrow \{\text{True}, \text{False}\}$ indicates for each pair of entities $\langle E_1, E_2 \rangle$ whether E_1 causes E_2 . This figure shows the maximum a posteriori estimate. At the top of the figure is the tree from domain D that is shared across all relational uses of D , where each leaf is an entity in the UMLS dataset. The first four figures at the bottom show four of the 49 relations modeled using Annotated Hierarchies. The green oval highlights the captured knowledge that chemicals cause diseases. The fifth figure shows just the causes relation, as modeled by the IRM. Note that the way the IRM does explain the causes data as well as the Annotated Hierarchies model (i.e., there are significantly more data components whose parameters must be fit using less data, and there are non-homogeneous components). This reflects the IRM’s assumption that the domain D is partitioned in the same way along both axes of all 49 relations. Reproduced from [54] (figure 3) with permission.

using an early version of BLAISE to manage this complexity. The memoization and transactional caching provided by the BLAISE virtual machine were key to achieving good inference performance, while the BLAISE SDK modeling framework allowed development effort to be used effectively. Roy reports:

Implementing the Annotated Hierarchies model was a matter of defining the state space, joint density, and stochastic moves; we wrote virtually no generic MCMC code.

~ Daniel Roy

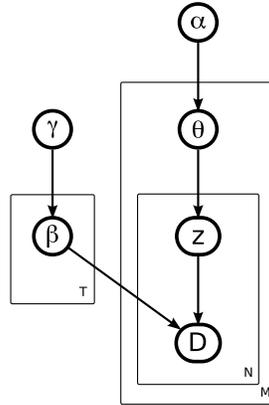


Figure 6-16: A Bayes net for the Latent Dirichlet Allocation model [5, 22] with M documents, each containing N words.

6.4 Latent Dirichlet Allocation

Author's role: Support

In the last few decades, technological advances have made it easy to instantly access vast numbers of natural language documents, provided that you can determine exactly which document you wish to access. The field of Information Retrieval focuses on indexing, organizing, and summarizing natural language corpora so that desired information can be located quickly. For example, the Google search engine is an information retrieval tool for documents on the internet. Topic modeling is a branch of information retrieval that seeks to organize documents into groups with similar semantic topics.

The Latent Dirichlet Allocation (LDA) model is a generative probabilistic model for topic modeling [5, 22]. LDA models topics as probability distributions over words. For a language with W words, each topic $\vec{\beta}_i$ is a W -element vector sampled from a symmetric Dirichlet distribution

$$\vec{\beta}_i \sim \text{DIRICHLET}(\gamma).$$

Each document \vec{D}_j is represented as a bag of words. Associated with each document is a probability distribution over topics, denoted $\vec{\theta}_j$. For an LDA model with N topics,

each $\vec{\theta}_j$ is a N -element vector sampled from a symmetric Dirichlet distribution

$$\vec{\theta}_j \sim \text{DIRICHLET}(\alpha).$$

Each word $\vec{D}_j[k]$ is generated by first sampling a topic $\vec{z}_j[k]$ using the document's distribution over topics

$$\vec{z}_j[k] \sim \text{MULTINOMIAL}(\vec{\theta}_j)$$

then sampling a word from that topic's distribution over words

$$\vec{D}_j[k] \sim \text{MULTINOMIAL}(\vec{\beta}_{\vec{z}_j[k]}).$$

Intuitively, each document in an LDA model has an effective distribution over words produced by a linear combination of the N topic-word distributions $\vec{\beta}_i$, weighted by the document-topic distribution $\vec{\theta}_j$. Thus the topic-word distribution vectors form a linear algebraic basis for the intuitive document-word distribution. Document likelihoods will be maximized when the document-word distributions most closely match the observed word frequencies in the documents. It is therefore the intuitive goal of inference to determine an appropriate basis set $\vec{\beta}_i$ from which to construct the document-word distributions. These basis vectors are probability distributions, so they can contain only positive values. It follows that the best vectors for the basis set will be those that put probability mass on words that are typically used in the same document – that is, words that are about the same topic of discussion.

Beau Cronin (MIT Brain and Cognitive Sciences, Navia Systems, Inc) designed and implemented LDA in BLAISE (see figure 6-17 for an implementation sketch). As a brief demonstration, the LDA implementation was used to perform topic analysis on the introductions of 32 Wikipedia [1] articles. Example input is seen in figure 6-18. The topics extracted with this model can be seen in figure 6-19.

Ontology:

Ontology is a study of conceptions of reality and the nature of being. In philosophy, ontology is the study of being or existence and forms the basic subject matter of metaphysics. It seeks to describe or posit the basic categories and relationships of being or existence to define entities and types of entities within its framework.

Some philosophers, notably of the Platonic school, contend that all nouns refer to entities. Other philosophers contend that some nouns do not name entities but provide a kind of shorthand way of referring to a collection (of either objects or events). In this latter view, mind, instead of referring to an entity, refers to a collection of mental events experienced by a person; society refers to a collection of persons with some shared interactions, and geometry refers to a collection of a specific kind of intellectual activity.

As a philosophical subject, ontology chiefly deals with the precise utilization of words as descriptors of entities or realities. Any ontology must give an account of which words refer to entities, which do not, why, and what categories result. When one applies this process to nouns such as electrons, energy, contract, happiness, time, truth, causality, and God, ontology becomes fundamental to many branches of philosophy

Reality:

Reality, in everyday usage, means “the state of things as they actually exist.” The term reality, in its widest sense, includes everything that is, whether or not it is observable or comprehensible. Reality in this sense may include both being and nothingness, whereas existence is often restricted to being (compare with nature).

In the strict sense of philosophy, there are levels or gradation to the nature and conception of reality. These levels include, from the most subjective to the most rigorous: phenomenological reality, truth, fact, and axiom.

Figure 6-18: The introductions to 32 Wikipedia [1] articles were used as documents to demonstrate Latent Dirichlet Allocation topic analysis. Shown here are 2 of the 32 documents. Stop words (such as: “the,” “an,” “to,” and “or”) were removed and a rudimentary word stemming was performed before analysis. Wikipedia articles are copyrighted by Wikipedia contributors and licensed under the GNU Free Documentation License; these excerpts are believed to be covered by fair use.

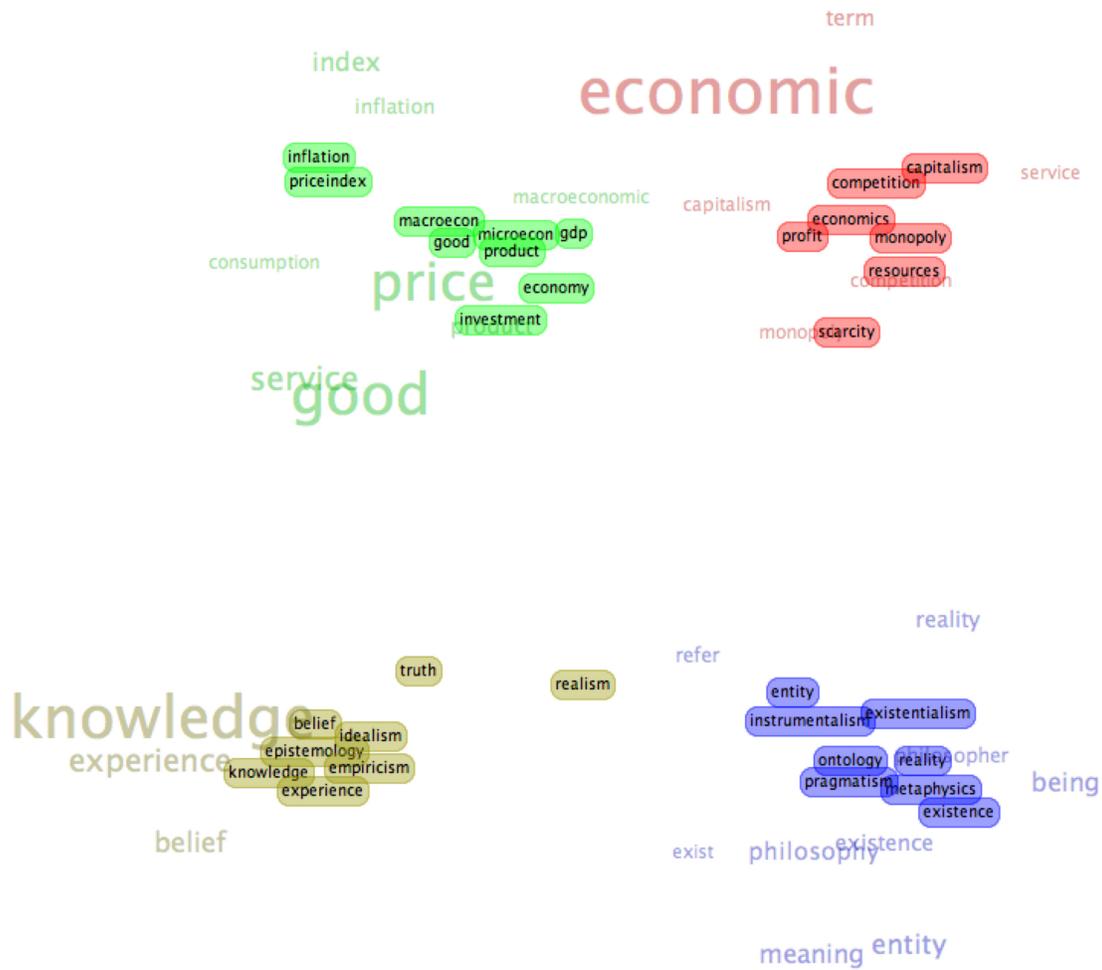


Figure 6-19: This figure depicts the topics LDA infers when applied to the dataset described in figure 6-18, assuming 4 topics. Nodes are documents, labeled by their Wikipedia title, colored by their predominant topic, and projected from the 4-dimensional topic simplex to a 2-dimensional space for visualization. The most common words in each topic are also displayed, with size proportional to frequency in the topic.

6.5 Standard Graphical Models in BLAISE

Author's role: Implementation Lead

A wide range of standard graphical modeling languages can be embedded in BLAISE. For example, factor graphs are particularly natural to represent. Recall that a factor graph is a bipartite graphical model with variables X as one type of node and factors F as the other. Each factor $F_j \in F$ is connected to a set of variables $X_j \subset X$ and has a function $f_j : X_j \rightarrow \mathbb{R}$ associated with it, such that the joint density of the graphical model is given by

$$p(X) \propto \prod_j f_j(X_j)$$

A simple example of a factor graph is the two-dimensional Ising model from statistical mechanics, in which the variables $X_{ij} \in \{1, -1\}$ form a grid of spins. Factors connect each adjacent pair of spins X_1, X_2 with the function $f(X_1, X_2) = e^{JX_1X_2}$, providing soft constraints that adjacent spins should be the same or different (corresponding to “ferromagnetic” or “antiferromagnetic” models and controlled by the “coupling” parameter J). Each spin may also have a unary factor $e^{HX_{ij}}$ attached to it, representing an external field preferring the spin to be positive or negative and controlled by the parameter H . Thus, the joint density for the model is

$$p(X) \propto \left(\prod_{\text{adjacent } X_1, X_2} e^{JX_1X_2} \right) \left(\prod_{X_{ij}} e^{HX_{ij}} \right).$$

A BLAISE model can be constructed for any factor graph. The BLAISE State hierarchy for the model is formed by using a State object for each variable, with the root of the model being a Collection State containing each of the variable States. Likewise, the BLAISE Density for the factor graph can be constructed by using a Density object for each factor, with each Density having State→Density edges to the variable States on which that factor depends. The root Density is a Multiplicative Collection Density containing each of the factor Densities.

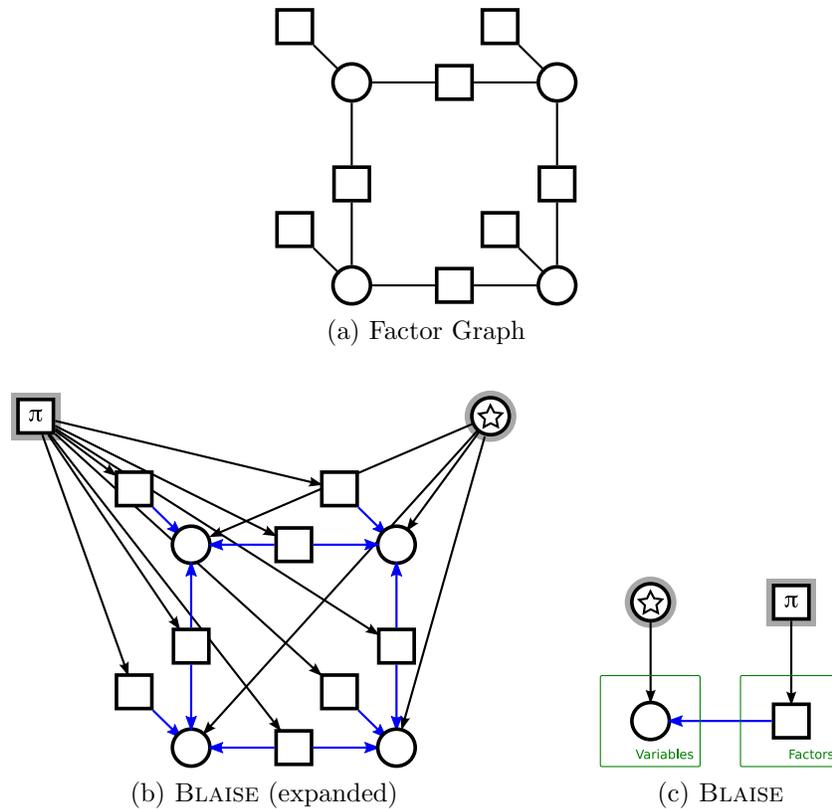


Figure 6-20: Lattice-structured factor graphs called Ising models are standard models from statistical mechanics (a). In BLAISE, any factor graph can be represented as a Collection State containing all the variables, plus a Collection Density containing all the factor densities (b),(c).

A variety of inference methods could be used, so there are many options for constructing the Kernel hierarchy; however, many of these can be easily automated. For example, if all the variables States are the same type, as in the Ising model, one could use a Virtual Hybrid Kernel on the Collection State, where the virtualized subkernel is a variable-type-appropriate single-site Metropolis-Hastings Kernel or enumerative Gibbs Kernel. If variables are of different types, the virtualized sub-Kernel could be a Conditional Hybrid Kernel which selects an appropriate Metropolis-Hastings or Gibbs sub-Kernel based on the type of the variable State. Of course, more complicated inference is also possible, using specialized inference Kernels for different States, or blocking the sampling of several variables together by using a Cycle Hybrid Kernel as the proposal of a Metropolis-Hastings or Gibbs Kernel.

Other graphical modeling languages, such as Bayes nets or Markov Random Fields

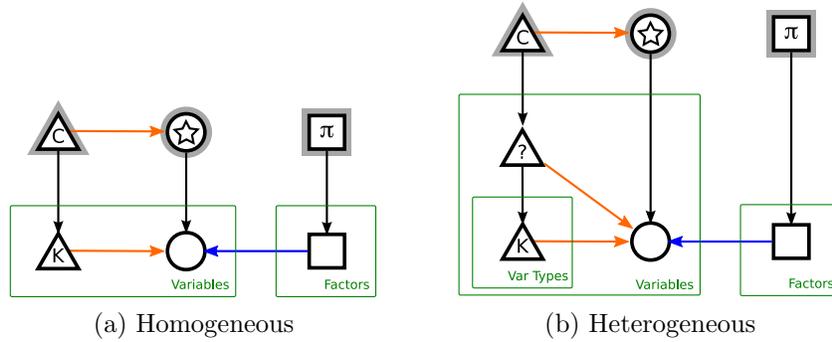


Figure 6-21: Kernels can be dispatched onto the variables in a factor graph using a virtual cycle Kernel (a). If the factor graph is heterogeneous – that is, it contains variables of different types – a conditional hybrid Kernel may be used to dispatch the appropriate Kernel for a variable’s type (b).

(MRF, i.e. undirected graphical model.), can also be embedded in BLAISE, because every Bayes Net or MRF can be reduced to a factor graph, which can then be embedded in BLAISE as described above.

The factor graph equivalent to a given Bayes Net will have the same set of variable nodes as the Bayes Net. In a Bayes Net, each variable V has a set of parent nodes $parents(V)$, where the parent nodes are exactly those nodes V_p for which a directed edge $V_p \rightarrow V$ exists. Each variable V also has a conditional probability function $p(V|parents(V))$, such that joint density on the Bayes Net is

$$p(\cdot) = \prod_V p(V|parents(V)).$$

This matches the joint density of a factor graph $p(\cdot) \propto \prod_j f_j(X_j)$ when, for each variable V , there is a factor f_V connected to V and its parents that evaluates the conditional probability function defined at V : $f_V(V, parents(V)) = p(V|parents(V))$.

Like Bayes Nets, the factor graph equivalent to a given Markov Random Field will have the same set of variable nodes as the MRF. In an MRF, each clique $V_{\{k\}}$ of variables has a potential function $\phi_k(V_{\{k\}})$ associated, such that the joint density on the MRF is

$$p(\cdot) \propto \prod_{V_{\{k\}}} \phi_k(V_{\{k\}})$$

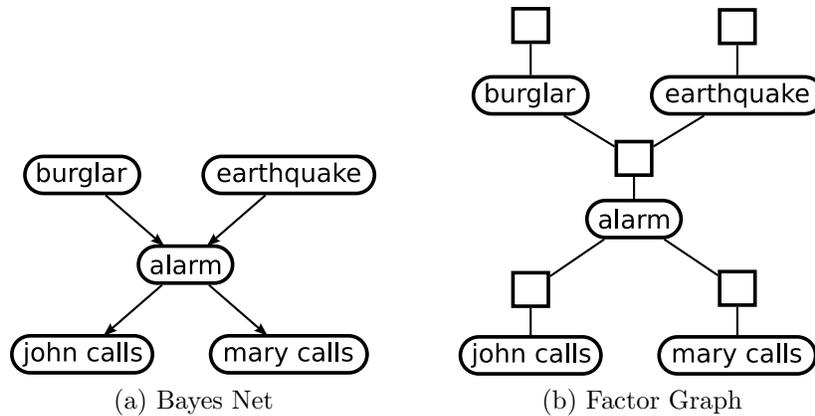


Figure 6-22: Any Bayes net can be reduced to a factor graph by converting each variable’s conditional probability function to a factor that evaluates that function, as seen in this figure with a simple Bayes net. Because BLAISE can model any factor graph, it can therefore also model any Bayes net.

This matches the joint density of a factor graph $p(\cdot) \propto \prod_j f_j(X_j)$ when, for each clique $V_{\{k\}}$, there is a factor $f_{\{k\}}$ connected to each node in the clique that evaluates the potential function for that clique: $f_{\{k\}}(V_{\{k\}}) = \phi_k(V_{\{k\}})$.

Factor graphs, Bayes Nets, and Markov Random Fields are simplistic uses of the BLAISE SDK modeling language; however, there is still significant advantage to be derived from BLAISE. All the transformations defined in chapter 4 are now easy to apply to models in any of these languages. For example, it is now a simple task to parallel temper a Bayes Net or use simulated annealing on a factor graph.

6.6 Systematic Stochastic Search

Author’s role: Collaborator

Classical search and probabilistic inference, two of the most fundamental algorithmic building blocks of artificial intelligence, share a variety of deep commonalities, yet are usually treated independently. Systematic stochastic search is a new research program that attempts to unify these two operations to produce new, more efficient algorithms.

Classical search algorithms (e.g. A*) and Monte Carlo probabilistic inference algorithms have related goals: locate high-scoring states in a state space. However,

the goals differ in an important way: classical search typically seeks to find the single best-scoring state, where Monte Carlo inference algorithms seek to produce states (i.e. samples) proportionally to their score, so that high-scoring states are sampled more frequently, but lower-scoring states should also be sampled, just less frequently.

Markov chain Monte Carlo, one of the most commonly used branches of Monte Carlo inference, also differs from classical search algorithms in how solution states are formed. Classical search algorithms such as A* typically start with an empty state which is extended step-by-step into a complete solution state. In contrast, MCMC starts with some complete, though potentially low-scoring, solution state and makes a sequence of small updates to the state. For example, if trying to find short paths through a map from point A to point B, a classical search algorithm would begin with a empty path starting and ending at A, and incrementally extend that path to a neighbor of A, then to a neighbor of that neighbor, and so on until the path reached B. In contrast, MCMC would start with some (probably highly suboptimal) path from A to B, and try making a series of small variations while always maintaining the path's endpoints.

In this regard, Markov chain Monte Carlo is more closely akin to constraint propagation algorithms, which start with a superset of the valid solutions and iteratively whittle away undesired states. In fact, probabilistic inference, particularly on factor graphs, can be viewed as soft constraint satisfaction; that is, constraints can be violated, but violating a constraint incurs a penalty proportional to the violation. Under this interpretation, the interest distribution for probabilistic inference encodes how well the constraints are being satisfied: well-satisfied constraints correspond to high probability states.

Classical search algorithms, on the other hand, seem more closely related to Monte Carlo methods based on importance sampling, because both start with empty solutions which are built up incrementally. Likelihood weighting for a Bayes net is the most direct analog, allowing a weighted sampled to be produced by visiting variables in topological order from parents to children. Each unobserved variable is sampled conditioned on its parents, and each observed variable is assigned the observed value,

adjusting the weight of the sample based on how likely it would have been to sample this value instead of just assigning it. Sampling unobserved variables corresponds to extending a path in classical search, and adjusting the sample weight to account for evidence corresponds to determining whether the goal has been reached (again, using a soft goal-function in the probabilistic setting rather than the hard binary goal function common in classical search).

Particle filtering² is no more than likelihood weighting applied to the specific setting of dynamic models; i.e. probabilistic models with structure that recurs for each time step in a dynamic process. Importance sampling methods suffer when variables that are sampled early in the process must take on *a priori* unlikely values in order to produce good solutions. Particle filters work around this difficulty by evolving a fixed number of particles (samples) in parallel, and allowing highly weighted particles to be replicated into multiple particles, while low weighted particles are killed off. In this way, particle filtering is closely related to beam search algorithms in classical search.

Systematic stochastic search [40] is a new research program that attempts to unify all of the techniques just described into a single search and inference framework. By unifying the previously disparate techniques, the hope is to transfer insights from classical search to probabilistic inference and vice versa, resulting in the construction of more efficient inference and search algorithms.

Systematic stochastic search rests on three insights:

1. Classical search and constraint problems can be viewed as the deterministic limit of probabilistic inference problems
2. MCMC updates can be run on partially constructed states, allowing MCMC updates to be interleaved with importance sampling extensions.
3. Particle filtering methods are applicable to all probabilistic models, not just dynamic models

Basic systematic stochastic search [40] enables particle filtering on any factor graph, visiting the variables in any pre-determined order³. In BLAISE, systematic

²at least, in the single-particle no-resampling setting

³Because the variables may be visited in any order, the algorithm designer is free to choose

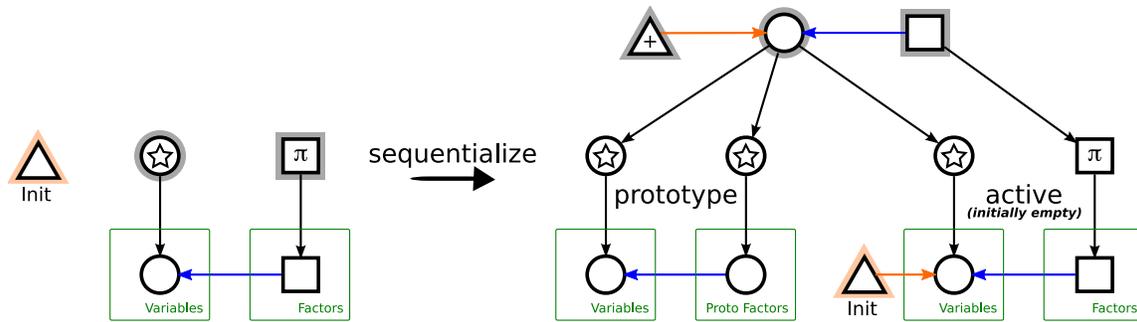


Figure 6-23: The sequentialization transform takes a complete factor graph and produces version of the same factor graph suitable for sequential Monte Carlo (e.g. particle filtering). The transform constructs an empty factor graph as the “active” factor graph. The transform also builds a “prototype” from the original factor graph. This prototype replaces all the original factor graph’s Densities with States so they may be manipulated (as will be described momentarily). The prototype allows an extension Kernel (labeled “+”) to find a variable that needs to be added to the active factor graph each time it is called. As variables are added to the active factor graph, the root Density locates any proto-factor for which the last-attached variable was just added to the active factor graph and adds the corresponding factor to the factor graph as well (note that the root Density evaluates only the active factor graph’s Density; it has a link to the prototype factor graph’s Density only in order to locate proto-factors.) This transform, concatenated with the particle filtering transform (section 4.3), provides the *basic systematic stochastic search* algorithmic recipe described in [40].

stochastic search is implemented as a sequentialization transform that converts a factor graph to a sequential model, as seen in figure 6-23. This sequential model can then be passed to the particle filtering transform (section 4.3), yielding a complete inference method.

An example of systematic stochastic search applied to stereo vision can be seen in figure 6-24. BLAISE enabled the experimental implementation of systematic stochastic search that produced these results. Running the experiments in this paper also demonstrated that BLAISE scales well; our largest inference runs included 8 particles on the dataset described in figure 6-24, resulting in a BLAISE model with approximately 500,000 States and 1,500,000 Densities.

When asked about BLAISE’s impact on this project, Roy reports:

For the Systematic Stochastic Search project we relied heavily on SDK

an order that is appropriate to the model in hand. Various choices of variable ordering and the advantages thereof are presented in [40].

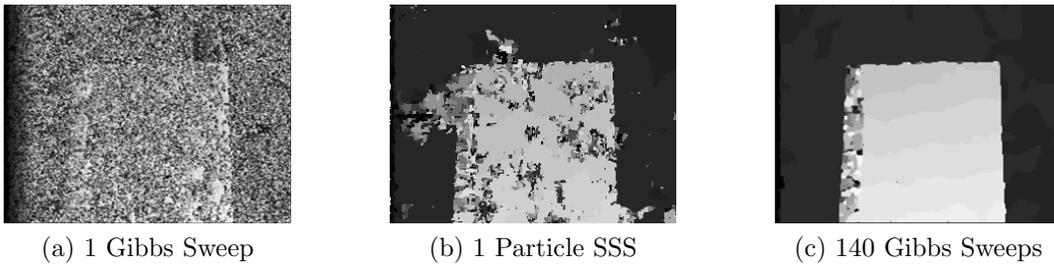
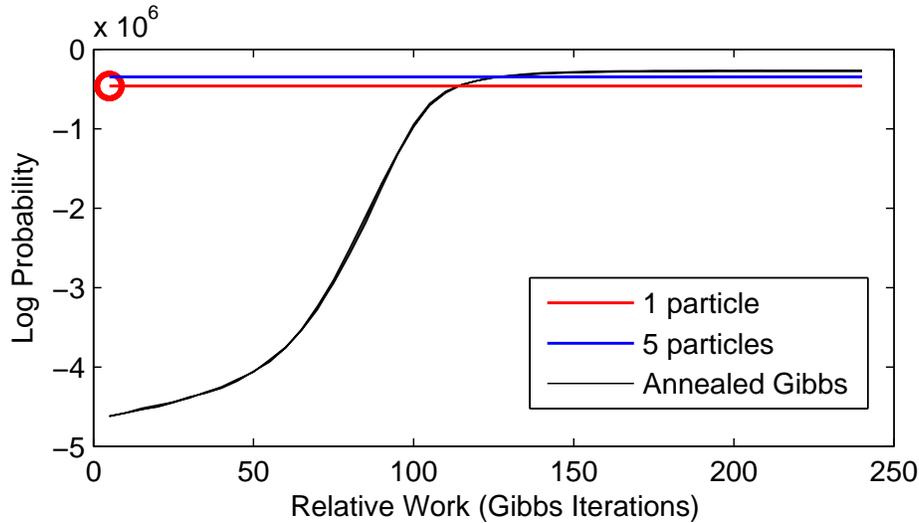


Figure 6-24: This figure compares the performance of annealed Gibbs sampling and the demonstration of the BLAISE-based implementation of systematic stochastic search [40] on a realistic stereo vision problem. The model was a 61,344 variable lattice-structured factor graph, with each variable having 30 possible values representing the possible stereo disparities at a particular image location. In this figure, the model was applied to the “map” image pair from the Middlebury Stereo Vision benchmark set. The main graph demonstrates that the systematic stochastic search method with a single particle reaches a high probably solution with significantly less computational effort than Gibbs sampling. (a) and (b) show the samples resulting from 1 Gibbs sweep and 1 particle systematic stochastic search, respectively. Though the computation time required for these two operations is equivalent, the result from systematic stochastic search is significantly better. For comparison, (c) shows the result obtained once the Gibbs sampler has converged after 140 sweeps.

transformations to re-express inference in factor graphs as systematic search. What made this efficient, despite its generality, was Blaise’s caching system.

~ Daniel Roy

Mansinghka describes his experience using BLAISE for this project:

SDKs and SDK transformations were a critical enabler for my research on systematic stochastic search. Without them, I doubt I would have understood how to implement search on generic factor graphs. Furthermore, without the Blaise Virtual Machine, I doubt I would have been able to conduct a broad range of satisfying experiments on the algorithm.

~ Vikash Mansinghka

6.7 Higher-level probabilistic programming languages

BLAISE SDK graphs are useful directly as a modeling language, but the BLAISE framework can also be viewed as an abstract machine upon which to construct even higher-level probabilistic modeling languages.

As we seek to build more and more sophisticated models, such higher-level languages will be necessary to manage the complexity. Sophisticated models will certainly require sophisticated inference techniques in order to perform inference tractably, and will most likely require special optimizing compilers in order to orchestrate this inference. Just as nearly all compilers go through multiple layers of intermediate representation, I hypothesize that compilers for high-level probabilistic languages will be no different. Because BLAISE reifies the elements of probabilistic inference methods into composable, reusable units, and inspired by the transformations presented in chapter 4, the BLAISE framework is uniquely well suited to become a foundation of higher level languages.

Model:

```
model {
  for (i in 1 : N) {
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i] * t[i]
    x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1)
  beta ~ dgamma(0.1, 1.0)
}
```

Data:

```
list(t = c(94.3, 15.7, 62.9, 126, 5.24, 31.4, 1.05, 1.05, 2.1, 10.5),
     x = c( 5, 1, 5, 14, 3, 19, 1, 1, 4, 22), N = 10)
```

Figure 6-25: “pump,” a standard BUGS [58, 62] model as it appears in the BUGS documentation’s examples. Recently, one week of development was sufficient to reimplement enough of BUGS on BLAISE to run this and many other standard BUGS models. BLAISE’s support for Constraint States (and automatic management of the long-distance dependencies they introduce) and standard library of reconfigurable inference components were critical elements in the rapid development of this project.

6.7.1 BUGS on BLAISE

Author’s role: Support

BUGS (**B**ayesian inference **U**sing **G**ibbs **S**ampling) [58, 62], is a popular system for MCMC inference on Bayes nets. BUGS consists of a declarative text-based modeling language for Bayes net models, together with software for automatically deriving a Monte Carlo inference algorithm based on the type of the full conditional distribution at each node. BUGS supports variables with domains of integers, real numbers or fixed-dimension matrices of these. BUGS variations have been in development since 1989. The ease with which BUGS models can be specified has lead to widespread adoption in the statistical community and pedagogical settings.

As an example of ease-of-development in BLAISE as well as BLAISE’s utility as a compilation target/interpreter substrate for other probabilistic languages, Beau Cronin (MIT Brain and Cognitive Sciences, Navia Systems, Inc) recently started reimplementing the BUGS language atop the BLAISE framework. As of this writing,

Parametric Model:

```
model {
  for (i in 1 : NComponents) {
    p[i] ~ dbeta(a, b)
  }
  w[] ~ ddirch(alpha[])
  for (j in 1 : NDatapoints) {
    z[j] ~ dcat(w[])
    x[j] ~ dbin(p[z[j]], OutOf)
  }
}
```

Parametric Data:

```
list(a = 1, b = 1,
     alpha = c(1, 1, 1), NComponents = 3,
     x = c(10, 12, 11, 13, 30, 32, 34, 33, 80, 78, 79),
     OutOf = 100, NDatapoints = 11)
```

Hypothetical Non-Parametric Model:

```
model {
  z[] ~ dcrp(alpha, NDatapoints)
  NComponents <- ndistinct(z[])
  for (i in 1 : NComponents) {
    p[i] ~ dbeta(a, b)
  }
  for (j in 1 : NDatapoints) {
    x[j] ~ dbin(p[z[j]], OutOf)
  }
}
```

Hypothetical Non-Parametric Data:

```
list(a = 1, b = 1,
     alpha = 1,
     x = c(10, 12, 11, 13, 30, 32, 34, 33, 80, 78, 79),
     OutOf = 100, NDatapoints = 11)
```

Figure 6-26: Reimplementing BUGS on the BLAISE framework should enable the language features of BUGS to be extended in a variety of useful directions. For example, extending the language to support for loops with non-constant bounds and including nonparametric distributions would allow nonparametric mixture models to be implemented in BUGS. This figure compares a traditional BUGS parametric mixture model (top) with a nonparametric mixture model in a hypothetical extended-BUGS language (bottom). Note that `NComponents` is part of the data in the parametric model, but is an inference target in the nonparametric model.

the project has received one week of development time, and already has sufficient coverage of the language that many standard BUGS models run without any modification (for example, see figure 6-25). A sizable portion of that time was spent creating a parser for the BUGS language specification. Beyond that, most of the implementation was assembled from standard BLAISE components. The implementation uses standard BLAISE States to represent variables. Deterministic assignments (“logical nodes” in BUGS terminology, e.g. `lambda[i] <- theta[i] * t[i]`) and math expressions are implemented as Constraint States, while stochastic assignments (e.g. `alpha ~ dexp(1)`) are implemented as Densities in the BLAISE model. Inference is implemented as a Concrete Mixture Kernel over standard BLAISE Kernels for each unobserved variable in the model. We hope that, in the future, we can also make more of BLAISE’s features available through extensions to the BUGS language. For example, `for` loops in BUGS are required to have their iteration bounds fixed at compile time; BLAISE’s support for transdimensional MCMC should allow us to remove this limitation. With this in place, it might also be possible to bring nonparametrics, such as the Chinese Restaurant Process, to BUGS models. Figure 6-26 shows what a nonparametric mixture model might look like in such an extended BUGS language.

6.7.2 Church: A stochastic lambda calculus

Author’s role: Collaborator

As a final demonstration of BLAISE’s expressivity, I present Church [20], a universal language for generative models with non-parametric memoization and approximate inference. Church builds on a pure (i.e. mutation-free) subset of Scheme [2]. Church adds *random procedures* such as: (`flip`) that flips a fair coin, (`flip weight`) that flips a weighted coin, (`normal mean var`) that draws a sample from a normal distribution, etc. Church also adds the primitive (`mem a p`) that returns a *stochastically memoized* version of the procedure `p` (see figure 6-27). As with any Lisp variant, Church has full support for higher-order procedures; for example, you can write (`lambda (f1 f2 x) ((if (flip) f1 f2) x)`) to define a procedure that takes two

Church IRM

```
(defmem 1.0 (drawclass domain) (gensym))
(defmem 0 (class entity domain) (drawclass domain))
(defmem 0 (component-mean class1 class2)
  (normal 0.0 10.0))
(defmem 0 (datapoint entity1 entity2)
  (normal (component-mean (class entity1 'T1)
                          (class entity2 'T2))
          1.0))
```

Figure 6-27: A complete Church [20] program for a two-domain ($T1 \times T2$) Infinite Relational Model [30] with normal-normal component models. In Church, the `(mem a p)` returns a stochastically memoized version of the procedure `p`, using a Chinese Restaurant Process with parameter `a` to govern whether a new draw from `p` is generated, or whether a previous draw from `p` is returned instead (or in Dirichlet Process terms, `p` provides the base measure for the DP). Note that invoking the memoized procedure with different parameters will access different Chinese Restaurant Processes. If `p` is a procedure taking a single parameter that has been memoized using `(define pmem (mem a p))`, then every evaluation of `(pmem 1)` will access the same CRP, but `(pmem 2)` will access a different one. This is akin to a traditional memoizer storing an independent value for each set of parameters a memoized procedure is called on. Because memoization is a very common operation in Church, syntactic sugar is provided to make it more concise. Analogous to the standard Scheme sugar where `(define (A args) B)` is syntactic sugar for `(define A (lambda (args) B))`, in Church `(defmem a (A args) B)` is syntactic sugar for `(define A (mem a (lambda (args) B)))`. Note that when `a` is 0, `mem` reduces to deterministic memoization. For example, in this program, `(defmem 0 (class entity domain) (drawclass domain))` ensures that a persistent class is assigned to each entity.

Church Infinite Hidden Markov Model

```
(defmem 1.0 (get-state) (gensym))
(defmem 1.0 (ihmm-transition state) (get-state))
(define (ihmm state length)
  (if (= n 0)
      '()
      (pair (pair state (observation-model state))
            (ihmm (ihmm-transition state) (- n 1)))))
```

Figure 6-28: This figure shows the implementation of an Infinite Hidden Markov Model [4] as a short Church program. Evaluating `(ihmm 'start 10)` will result in a sequence of 10 `(state . observation)` pairs sampled from the model, starting from the state `'start`.

single-argument procedures `f1` and `f2`, flips a fair coin to choose one of them, then applies it to `x`. The expressiveness of Church allows sophisticated probabilistic models to be expressed succinctly. For example, figure 6-27 shows how the Infinite Relational Model [30] (see section 6.3.1) can be written with just a handful of Church statements, and figure 6-28 shows the simple church program for an Infinite Hidden Markov Model [4].

Evaluation in Church equates to sampling; that is, `(eval expr)` draws a sample from the generative model defined by `expr`. Church also provides a new procedure `(query expr pred)`, where `pred` is of the form `(lambda (x) ...)` \mapsto `{True, False}`. Evaluating `(query expr pred)` draws a sample `s` from the generative model `expr`, conditioned on `(pred s)` evaluating to `True`. Note that `(query expr (lambda (x) True))` is equivalent to `(eval expr)`.

As part of Church's debut, we created a BLAISE-based MCMC algorithm for inference on Church models. The state space for this search is the space of all possible execution histories for a Church program. Elements of a search history (e.g. procedure applications, evaluations, random choices made by `flip`, environments, etc) are represented as different types of BLAISE States, with random choices also having appropriate Densities attached to them. MCMC inference is implemented by using Metropolis-Hastings Kernels to consider resampling a new value for random choices. Constraint State functionality is used extensively throughout the Church sys-

tem; whenever a value is changed (for example, if MCMC changes the outcome of a `(flip)`), any expression depending on that value receives a constraint message triggering it to re-evaluate itself (which will likely trigger further cascading re-evaluations). These re-evaluation cascades can also cause the state space to change dimensionality. For example, consider `(if (flip) 1 (normal 1 2))`. When the value of `(flip)` is `True`, only the consequent (i.e. `1`) is evaluated. If a Metropolis-Hastings proposal changes the value of `(flip)` to `False`, then only the alternate (i.e. `(normal 1 2)`) is evaluated. Because `normal` is a random procedure, this introduces new random state into the system. Church therefore also makes extensive use of Initialization Kernels to manage sampling values for these state-space dimensionality changes. It is striking that BLAISE’s long-distance compositionality patterns (i.e. Constraint States and Initialization Kernels) allow Church to use the stock BLAISE Kernels for changing the value of a random procedure. To reiterate, Church was implemented in BLAISE without writing a single custom inference Kernel, a testament to the power of composable inference.

In response to his experience using BLAISE to implement Church, Mansinghka writes:

Managing the complexity of a random walk over consistent execution histories of a Lisp machine would have been impossible without the constraints provided by the SDK language, which routinely exposed fundamental errors in our thinking.

~ Vikash Mansinghka

6.8 Discussion

In this chapter, I presented a variety of models implemented in BLAISE. These implementations demonstrate the breadth of models to which BLAISE is amenable. No matter how sophisticated or simple the model, BLAISE enabled the models to be implemented more rapidly and reliably. The BLAISE SDK modeling language has also proven itself to be extremely useful to modelers just in thinking about and discussing their models before implementation has even begun. For example, Roy reports:

The decomposition of Monte Carlo algorithms into abstract states, densities, kernels and transformations thereof simplifies a vast literature. At the same time, the composition rules that SDKs satisfy support the engineering of complex models. More than a few times during the Church project, apparent violations of these rules revealed mistakes in our preliminary sketches for universal inference.

~ Daniel Roy

The excitement of the user community around BLAISE attests to the framework's usefulness.

Chapter 7

Related Work

There are a variety of existing software packages for inference in probabilistic models. In this section, I briefly review some of the most popular tools¹.

7.1 Bayes Net Toolbox

The **Bayes Net Toolbox** for Matlab (BNT) [47] supports a variety of inference methods on Bayes nets. BNT supports variables with domains of integers, real numbers, or booleans. Conditional probability distributions for continuous variables are limited to Gaussian distributions; discrete variables typically use table-based distributions, though a few other distributions such as noisy-or or softmax are also available. BNT supports a wide variety of exact inference methods; for approximate inference, it supports belief propagation, likelihood weighting, and gibbs sampling (for discrete nodes only).

In contrast to BLAISE, BNT does not focus on Monte Carlo methods, nor does it support sophisticated models: there is no support for structured variable domains, non-parametric (or even commonly used parametric) distributions, nor models with unknown dimensionality.

¹Appendix B of [33] indexes a wider variety of Bayes net software packages.

	BNT	BUGS	VIBES	BLOG	IBAL	BLAISE
Continuous Domains	Yes	Yes	Yes	No	No	Yes
Structured Domains	No	No	No	Yes	Yes	Yes
Unknown Dimensionality	No	No	No	\pm NP ^a	-NP	+NP
Standard Distributions	No	Yes	No	Yes	Yes	Yes
Inference Scheme	E,O	MC	V	MC	E	MC
Advanced Inference	No	No	No	No	No	Yes
Compositional Inference	No	No	No	No	No	Yes
User-defined Inference	No	No	No	No	No	Yes

Figure 7-1: Existing software packages for probabilistic inference have different focuses than BLAISE.

- Continuous Domains: Are continuous domains supported? Yes is the preferred value.
- Structured Domains: Are structured domains (e.g. trees, graphs, etc.) supported? Yes is the preferred value.
- Models with unknown dimensionality: +NP = Yes, including nonparametrics; -NP = Yes, but not nonparametrics; No. +NP is the preferred value.
- Standard distributions: Yes = most common distributions are easily available and applicable anywhere in the model; No = there are serious restrictions on distribution choice. Yes is the preferred value.
- Inference Scheme: E=Exact, MC=Monte Carlo approximate, V=Variational approximate, O=other approximate. MC is the preferred value; see sections 2.1 and 2.2.
- Advanced Inference: Does the package support higher-order variations of inference methods (e.g., tempered inference). Yes is the preferred value.
- Compositional Inference: Does the package focus on composition of inference methods? Yes is the preferred value.
- User-defined Inference: Does the package encourage user-defined inference? Yes is the preferred value.

^aThe NP-BLOG extension to BLOG aims to support nonparametrics, but this support is not a standard part of BLOG.

7.2 BUGS

BUGS (**B**ayesian inference **U**sing **G**ibbs **S**ampling) [58, 62], is a popular system for MCMC inference on Bayes nets. BUGS supports model creation in either a text-based modeling language or a graphical Bayes net editor called DoodleBUGS. BUGS supports variables with domains of integers, real numbers or fixed-dimension matrices of these. Given a BUGS model, the software will automatically derive a Monte Carlo inference algorithm based on the type of the full conditional distribution at each node.

In contrast to BLAISE, BUGS does not support user-defined or structured variable domains, nor does BUGS allow models of unknown dimensionality², so sophisticated models including such structures as trees or graphs are not supported. While BUGS's automatically derived inference is good for many relatively simple models, BUGS does not have much support for tailoring inference to take advantage of the domain structure.

7.3 VIBES

VIBES (**V**ariational **I**nference in **B**ay**ES**ian networks) [66] is a variational message passing system with modeling features similar to BUGS. VIBES models are defined using a graphical Bayes net editor or by writing XML files. Variational message passing requires distributions to be conjugate around each variable in the model; while it may be possible to use Monte Carlo approximations for variables with non-conjugate distributions, this is not implemented.

In contrast to BLAISE, VIBES can handle only a very restricted class of models. In addition to supporting only a few types of conditional distributions, and the constraint of having to satisfy conjugacy, VIBES does not support user-defined or structured variable domains, nor does it allow models of unknown dimensionality, so sophisticated models including such structures as trees or graphs are not supported.

²There exists an experimental Reversible Jump extension to WinBUGS [36]. However, even with this extension, only a limited class of transdimensional models are supported.

7.4 BLOG: Bayesian Logic

BLOG (**B**ayesian **L**ogic) [45] is a language for describing probabilistic models over relational structures, where the number of objects in the model is unknown and object identity may be uncertain. Models over sophisticated domains can be expressed using the relational structures. While BLOG does not include nonparametrics, a non-parametric extension (NP-BLOG) has been proposed [8]. [45] also provides inference over BLOG models, using likelihood weighting and Markov chain Monte Carlo.

In contrast to BLAISE, BLOG focuses largely on the declarative specification of probabilistic models, whereas BLAISE focuses on the composition of inference. BLAISE also provides a variety of sophisticated inference techniques that are not available in the BLOG. It could be interesting future work to use BLOG (or NP-BLOG) as a modeling language for building BLAISE models.

7.5 IBAL

IBAL [51] (**I**ntegrated **B**ayesian **A**gent **L**anguage) is a functional language with stochastic choice primitives, designed for describing probabilistic models as a generative process. IBAL models sophisticated domains naturally, using standard functional programming representations (e.g., lists formed from “cons” cells). Variables may be integers, booleans, symbols, or abstract data types over these; although real numbers are not primitive types in IBAL, [50] shows how they may be implemented using IBAL’s abstract data types.

In contrast to BLAISE, IBAL uses exact inference methods; as a result, as models grow more sophisticated, they are likely to become intractable in IBAL’s inference scheme. Implementing a Markov chain Monte Carlo version of IBAL would require a significant departure from IBAL’s current representational assumptions.

Chapter 8

Conclusion

My thesis is that a framework for probabilistic inference can be designed that enables efficient composition of both models and inference procedures, that is suited to the representational needs of emerging classes of probabilistic models, and that supports recent advances in inference.

I have supported this thesis by developing BLAISE, including the BLAISE State-Density-Kernel graphical modeling language, BLAISE transformations, and BLAISE virtual machine, and by means of several sophisticated applications built on BLAISE. In this chapter, I outline how BLAISE enables new perspectives and future research, followed by a review of the contributions I have made to the field of artificial intelligence.

8.1 New perspectives

The BLAISE modeling language provides a more unified view of Monte Carlo inference by making inference choices explicit in a human-readable graphical modeling language. For example, many MCMC practitioners conflate single-site Gibbs kernels with a Gibbs inference sweep; explicit cycle kernels make this distinction clear. Others fail to distinguish a mixture of Metropolis-Hastings kernels from a single M-H kernel with a mixture of proposals; likewise, for both M-H and Gibbs sampling, the

notion and impact of blocking variables is frequently overlooked. BLAISE makes these considerations tangible, as different arrangements of the same components (e.g. a M-H Kernel and a Cycle Kernel) in the BLAISE SDK graph (e.g. figures 3-18 and 3-19). At the most fundamental level, many practitioners do not even think in terms of composable Kernels, with the result that effective inference methods are either overlooked or used without any justification (e.g. mixing both Gibbs sampling and other Metropolis-Hastings methods within the same inference task).

BLAISE transformations also provide a critical avenue for unifying our understanding of sophisticated probabilistic inference, by making explicit exactly how an existing model/inference routine should be adapted to achieve such inference, using a language with sufficient precision that these changes can be completely automated. Precise, compact descriptions significantly lower the barrier of entry to working with these techniques, while having the descriptions phrased in terms of reusable pieces encourages the exploration of novel variants. For example, interpreting parallel tempering as a BLAISE transform (section 4.2.3) makes the technique easy to use (simply extend the model using the transform), easy to analyze (Metropolis-Hastings-based swaps and composition using standard hybrid kernels guarantee the correct stationary distribution), and easy to extend (for example, by swapping only part of each chain’s state or tempering only part of each chain’s density).

By enabling new perspectives such as these, BLAISE holds the potential to transform the perception of Monte Carlo inference from a hodgepodge of isolated techniques into coherent engineering discipline.

8.2 Enabled research

BLAISE is a critical component of a larger stack of software and hardware tools for high-performance, easy-to-use probabilistic inference. As part of this thesis, I created the BLAISE SDK graphical modeling language, which enabled the creation of higher-level modeling tools such as Stochastic Lambda Calculus (section 6.7.2) and a reimplementa-tion of the popular BUGS language (section 6.7.1). Building on these

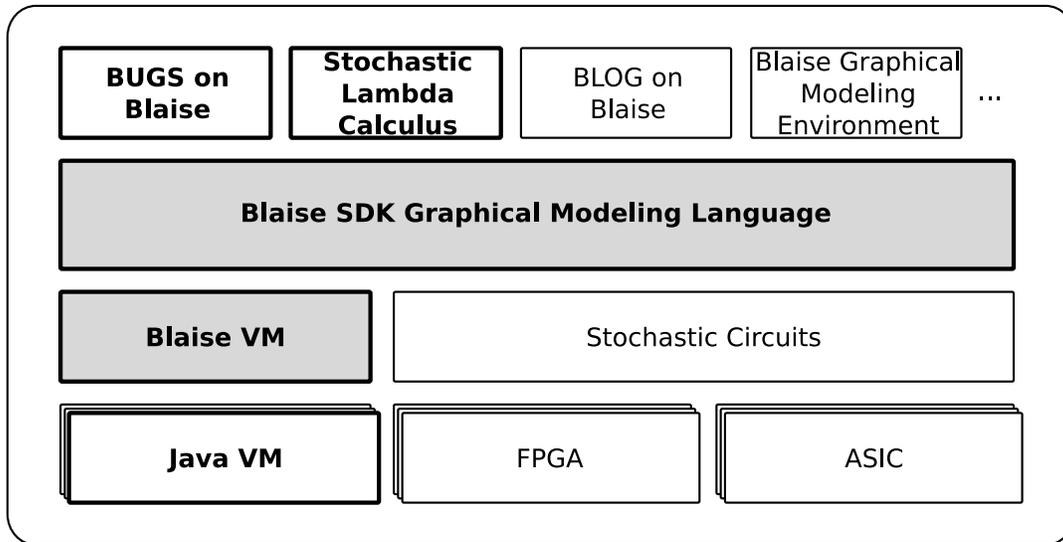


Figure 8-1: BLAISE is part of a larger stack of software and hardware abstractions. At the top of this stack are high level probabilistic modeling tools, such as BUGS on BLAISE (section 6.7.1) Stochastic Lambda Calculus (section 6.7.2), BLOG on BLAISE or a purely graphical modeling environment. All of these tools are implemented atop the BLAISE SDK language. I have already developed a Java-based BLAISE virtual machine to execute SDK models, but other execution environments are also possible, including custom built stochastic circuits. In this figure, the parts of the abstraction stack that I developed and that were central to this thesis are shaded gray, with supporting elements discussed as part of this thesis in bold.

successes, I am also exploring an implementation of BLOG (**B**ayesian **L**ogic) [45], a first-order probabilistic modeling language, atop the BLAISE modeling language. In addition, given the range of models that can be created by mixing and matching standard BLAISE SDK elements, it should be possible to create a point-and-click graphical modeling environment for BLAISE that allows the user to draw SDK diagrams on screen, apply transformations with a click, and execute the resulting model on the BLAISE virtual machine. With these tools, it would be possible to create even complicated models in a matter of hours, rather than the weeks or months currently standard. Furthermore, because BLAISE is designed for extensibility, it should be relatively easy for users to customize inference methods to suit the model, or even create entirely new SDK elements to interact with these high-level tools.

In this thesis, I also invented the BLAISE virtual machine to execute BLAISE SDK graphs efficiently on common off-the-shelf hardware. However, this is only one pos-

sible execution environment. MIT researchers Vikash Mansinghka and Eric Jonas are currently developing a suite of stochastic circuit primitives to exploit hardware-level parallelism on field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). Stochastic circuit implementations of Monte Carlo algorithms can produce massive increases in speed – sometimes even converting linear time algorithms to constant time algorithms. I am working with Mansinghka and Jonas to use stochastic circuits as an alternative to the BLAISE virtual machine. This research will focus on developing a compiler for SDK graphs that can target the stochastic circuit machines. The BLAISE SDK language is well-suited for this purpose because it makes inference into an explicit and manipulable element of the model, enabling a compiler to interpret it.

Trends in computing hardware today indicate that parallelism will be an important aspect of high-performance software, even without customized hardware. Large compute clusters are becoming more commonplace in both commercial and academic settings, and even personal computers are typically have 2–8 processor cores today. It follows that another important avenue of future research for BLAISE is automatic parallelization. For example, a future version of BLAISE might support a “parallel hybrid Kernel” that operates somewhat like a cycle Kernel, but makes no guarantees about the order in which its child Kernels are applied. On a serial machine, a parallel hybrid Kernel would pick an arbitrary order in which to execute its child Kernels, but on a parallel machine, it might execute several of its child Kernels simultaneously on different processing units. So long as the child Kernels operate on conditionally independent portions of the State–Density graph, the results on the parallel machine should be indistinguishable from the results on a serial machine.

Automatic optimization of inference in a BLAISE model is another exciting avenue of future research. For example, detecting that a cycle Kernel can be safely converted to a parallel hybrid Kernel could result in dramatic performance increases with no effort from the modeler. As mentioned in chapter 4, other transformations might also be automatically applied as well, such as conjugacy-exploiting transformations, the parallel tempering transformation, or other inference-enhancing transformations

yet to be designed. Furthermore, automatic optimization has the advantage that, if the modeler later makes changes that prevent a particular optimization strategy, the inference may get slower, but the modeler does not need to completely reimplement her model to make it functional again.

The most exciting aspect of these potential research paths is their SDK-mediated interaction – advances in any of these research paths bring more power to all the others. How long will it be before we have automatically parallelized parallel-tempered stochastic lambda calculus models running on thousand node supercomputers or simulated annealed BLOG models running on a custom-purpose ASIC? Only time will tell.

8.3 Contributions

In this thesis I have made several contributions to the field of artificial intelligence.

I began by identifying probabilistic models as a key component both to understanding human-like cognition and to addressing industrial data interpretation challenges, after which I identified several key shortcomings in the tools we currently use when discussing and implementing probabilistic models. Chief among these concerns was that our current approaches to probabilistic inference do not compose in the same way that probabilistic models do; though a modeler might be able to compose two traditional models together with only moderate work, she would almost certainly be forced to reimplement inference for the composite model from scratch. Implementing probabilistic inference is a time consuming and error-prone process, and sophisticated inference techniques are typically eschewed due to the complexity they would add to the system.

With these thoughts in mind, I introduced my thesis statement:

My thesis is that a framework for probabilistic inference can be designed that enables efficient composition of both models and inference procedures, that is suited to the representational needs of emerging classes of probabilistic models, and that supports recent advances in inference.

I then began to address this statement by inventing the BLAISE State–Density–Kernel (SDK) graphical modeling language. The SDK language is distinctive in that it explicitly represents in the graphical model both inference (via Kernels) and composition (via State, Density, and Kernel hierarchies). I also created a number of novel composition strategies, including Constraint States, non-linear Densities, virtual hybrid Kernels, conditional hybrid Kernels, and Initialization Kernels.

Using this modeling language, I reinterpreted a variety of existing probabilistic inference techniques, including Gibbs sampling, Metropolis-Hastings, simulated annealing, parallel tempering and particle filtering, as straightforward transformations of a BLAISE SDK model. In contrast to the unmanageable complexity traditionally introduced by these methods, in BLAISE these transformations are point-and-click/one-line-of-code operations and may even be automatable in the future. As a concrete example of the flexibility obtained by compositionality, I demonstrated how BLAISE transforms could isolate the choice of mixture model style from the implementation of mixture model components. This isolation allows the same component model to be used for mixtures varying from fixed-weight, fixed-size mixtures through non-parametric mixture models and even through infinite relational models, noting that the modeler could continue to use exactly the same inference implementation in all these cases.

Next, I implemented the BLAISE virtual machine, a Java-based software system running on common off-the-shelf hardware that can execute the stochastic automata represented by BLAISE SDK graphs and can perform BLAISE model transformations. In my implementation, I strove for efficiency, and have reported my conclusions regarding algorithms and data structures necessary for an efficient virtual machine implementation.

Finally, I validated the BLAISE framework through the implementation of several models and other probabilistic modeling languages atop the BLAISE SDK abstractions. These models included generative models of vision*, models for analysis of

*Author's role: Implementation Lead

neurophysiological data[†], models for relational data (including the Infinite Relational Model* [30] and the Annotated Hierarchies model[†] [54]), the Latent Dirichlet Allocation topic model[†] [5], traditional graphical models (Bayes nets, factor graphs)*, a reimplementaion of the BUGS [58, 62] probabilistic modeling language on BLAISE[†], and Stochastic Lambda Calculus probabilistic modeling language[‡]. In order to determine whether BLAISE was a productive and natural probabilistic modeling framework, I encouraged and supported other researchers in using BLAISE in their work, with great success, resulting in several of the applications above (my role in each application is noted with a footnote symbol.) I also demonstrated BLAISE’s potential to drive research in the development of probabilistic inference methods by collaborating on Systematic Stochastic Search[‡], a new research program that attempts to unify classical search and probabilistic inference, that has already resulted in a novel BLAISE-based transformation that generalizes particle filtering from dynamic Bayes nets to arbitrary sequentializations of factor graphs.

The ideas developed in this thesis are already changing the way people think about probabilistic inference, and enabling new perspectives and serving as the foundation for future work. For example, one user writes:

The SDK language has changed the way I think about probabilistic models, stochastic processes, and stochastic (and deterministic) state machines, by allowing me to build up all these objects out of pieces. It has also proved beautifully consistent with the constraints imposed by physical circuit design, bridging much of the gap between the abstract descriptions of stochastic processes from Church and the messy world of registers, combinational logic, and state transitions. I am excited to continue working with the language, both as an intermediate representation in a series of linked compilers and as a source of inspiration for new models of computation.

~ Vikash Mansinghka

The foundation for each of these contributions is a single idea: composable probabilistic inference. This is the key to building sophisticated probabilistic models tractably, and BLAISE shows that it is possible today.

[†] Author’s role: Support

[‡] Author’s role: Collaborator

Appendix A

Conditional Hybrid Kernels

Conditional Hybrid Kernels are a new class of MCMC hybrid kernels that implement a conditional control flow semantics for inference Kernels, much as `if` statements or `case` statements do in programming languages.

A.1 An Introduction to Conditional Hybrid Kernels

The intuition behind Conditional Hybrid Kernels is simple: a predicate partitions the state space into two subspaces, and a different subkernel is applied for states in each subspace. However, even if one assumes that both subkernels have the desired stationary distribution, care must be taken to ensure that the resultant Conditional Hybrid Kernel has the same stationary distribution. For example, consider using MCMC to explore the very simple distribution $p_{target}(x) = Binomial(x; 3, 0.5) = \begin{bmatrix} 0.125 & 0.375 & 0.375 & 0.125 \end{bmatrix}$ (the probability of flipping three fair coins and getting x heads). Transition kernels for this space can be compactly represented as 4x4 matrices. Many transition matrices have the appropriate stationary distribution; consider the following two arbitrarily selected transition matrices, each with p_{target} as

a stationary distribution:

$$K_1 = \begin{bmatrix} 0.265 & 0.249 & 0.229 & 0.257 \\ 0.083 & 0.709 & 0.204 & 0.004 \\ 0.076 & 0.204 & 0.624 & 0.095 \\ 0.257 & 0.011 & 0.285 & 0.447 \end{bmatrix} \quad K_2 = \begin{bmatrix} 0.666 & 0.041 & 0.099 & 0.194 \\ 0.014 & 0.864 & 0.121 & 0.002 \\ 0.033 & 0.121 & 0.835 & 0.011 \\ 0.194 & 0.004 & 0.032 & 0.770 \end{bmatrix}$$

(where state distribution $p_{x_{i+1}}$ equals $p_{x_i}K$).

If one partitions the state space into $\Omega_1 = \{0, 1\}$, $\Omega_2 = \{2, 3\}$, then a naïve attempt at a Conditional Hybrid Kernel using K_1 on Ω_1 and K_2 on Ω_2 would append the first two rows of K_1 to the last two rows of K_2 :

$$K_{naive} = \begin{bmatrix} 0.265 & 0.249 & 0.229 & 0.257 \\ 0.083 & 0.709 & 0.204 & 0.004 \\ 0.033 & 0.121 & 0.835 & 0.011 \\ 0.194 & 0.004 & 0.032 & 0.770 \end{bmatrix}$$

Unfortunately, K_{naive} no longer has the appropriate stationary distribution:

$$\begin{bmatrix} 0.125 & 0.375 & 0.375 & 0.125 \end{bmatrix} * K_{naive} = \begin{bmatrix} 0.101 & 0.343 & 0.422 & 0.138 \end{bmatrix}$$

The problem is that the transitions between partitions Ω_1 and Ω_2 are no longer balanced according to the target distribution. To resolve this issue, some additional constraint is needed. The solution presented here is simple: when constructing a Conditional Hybrid Kernel for a given partitioning of the state space, the subkernels must not generate transitions that cross the partition. More formally, I will constrain Ω_1, Ω_2, K_1 , and K_2 such that:

$$\begin{aligned} \forall x \in \Omega_1, x^* \in \Omega_2 : K_1(x \rightarrow x^*) = 0, K_2(x \rightarrow x^*) = 0 \\ \forall x \in \Omega_2, x^* \in \Omega_1 : K_1(x \rightarrow x^*) = 0, K_2(x \rightarrow x^*) = 0 \end{aligned} \tag{A.1}$$

In the coin flipping example, the constraint implies blocks of zeroes in certain

regions of K_1 and K_2 , so that both these subkernels and the resulting hybrid K_{Ω_1, Ω_2} assign zero probability to any partition crossing transitions:

$$K_1, K_2, K_{\Omega_1, \Omega_2} \in \left[\begin{array}{cc|cc} \cdot & \cdot & 0 & 0 \\ \cdot & \cdot & 0 & 0 \\ \hline 0 & 0 & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot \end{array} \right]$$

It is proven below that this constraint guarantees that the correct stationary distribution is produced. Furthermore, it is a constraint that is naturally satisfied in many situations. For example, in any multi-variable system, a single-site transition at variable X will never produce a partition-crossing transition for any partition function defined on variables other than X .

One consequence of this constraint is that Conditional Hybrid Kernels are never ergodic on their own, because Conditional Hybrid Kernels are not irreducible. Therefore, Conditional Hybrid Kernels will typically be used as subkernels for mixture hybrids or cycle hybrids.

For simplicity, this appendix focuses on Conditional Hybrid Kernels based on binary partitions of the state space. This is without loss of generality, because an n -ary partitioning of the state space can be reduced to a series of binary partitions, each carving out one piece of the n -ary partition and recursing to another binary Conditional Hybrid Kernel to handle the remaining $n - 1$ partitions.

A.2 Conditional Hybrid Kernel Stationary Distribution Proof

This section first formally defines a Conditional Hybrid Kernel in terms of its subkernels, and then uses that definition to prove that the stationary distribution of the Conditional Hybrid Kernel is a linear combination of the subkernels' truncated stationary distributions.

If Ω_1 and Ω_2 are disjoint state spaces, such that $\Omega = \Omega_1 \cup \Omega_2$, and if $K_1(x \rightarrow x^*)$

and $K_2(x \rightarrow x^*)$ are two transition kernels defined on Ω that satisfy equation A.1, then define $\hat{K}(x \rightarrow x^*)$, $x \in \Omega$, the Conditional Hybrid Kernel of K_1 and K_2 , as:

$$\hat{K}(x \rightarrow x^*) = \begin{cases} K_1(x \rightarrow x^*), & \text{if } x \in \Omega_1; \\ K_2(x \rightarrow x^*), & \text{if } x \in \Omega_2; \end{cases} \quad (\text{A.2})$$

Conditional Hybrid Stationary Distribution. *If K_1 has stationary distribution $p_1(x)$ and K_2 has stationary distribution $p_2(x)$, then for any $\alpha, 0 \leq \alpha \leq 1$, the Conditional Hybrid Kernel $\hat{K}(x \rightarrow x^*)$ has $\hat{p} = \alpha\hat{p}_1 + (1 - \alpha)\hat{p}_2$ as a stationary distribution, where \hat{p}_1 and \hat{p}_2 are renormalized truncations of p_1 and p_2 to Ω_1 and Ω_2 , respectively.*

Proof. I wish to prove that \hat{p} is a stationary distribution of $\hat{K}(x \rightarrow x^*)$; that is, \hat{p} is a valid distribution for which:

$$\hat{p}(x^*) = \int_{\Omega} \hat{p}(x) \hat{K}(x \rightarrow x^*) dx \quad (\text{A.3})$$

Because p_1 and p_2 are stationary distributions of K_1 and K_2 respectively, one can prove that \hat{p}_1 and \hat{p}_2 are also stationary distributions of K_1 and K_2 respectively. (See Truncated Stationary Distribution Lemma in the section A.3.) Therefore:

$$\hat{p}_1(x^*) = \int_{\Omega} \hat{p}_1(x) K_1(x \rightarrow x^*) dx \quad (\text{A.4})$$

$$\hat{p}_2(x^*) = \int_{\Omega} \hat{p}_2(x) K_2(x \rightarrow x^*) dx \quad (\text{A.5})$$

Expanding equation A.3 with $\hat{p} = \alpha\hat{p}_1 + (1 - \alpha)\hat{p}_2$ and noting that $\hat{K}(x \rightarrow x^*) = K_1(x \rightarrow x^*)$ everywhere that $\hat{p}_1(x)$ is nonzero, and likewise $\hat{K}(x \rightarrow x^*) = K_2(x \rightarrow x^*)$ everywhere that $\hat{p}_2(x)$ is nonzero, yields:

$$\alpha\hat{p}_1(x^*) + (1 - \alpha)\hat{p}_2(x^*) = \alpha \int_{\Omega} \hat{p}_1(x) K_1(x \rightarrow x^*) dx + (1 - \alpha) \int_{\Omega} \hat{p}_2(x) K_2(x \rightarrow x^*) dx \quad (\text{A.6})$$

which clearly holds given equations A.4 and A.5.

All that remains is to show that \hat{p} is a valid distribution; that is, $\int_{\Omega} \hat{p}(x) dx = 1$

and $\forall x \in \Omega : \hat{p}(x) \geq 0$. Because \hat{p}_1 and \hat{p}_2 are renormalized, $\int_{\Omega} \hat{p}_1(x) dx = 1$ and $\int_{\Omega} \hat{p}_2(x) dx = 1$. Therefore $\int_{\Omega} \hat{p}(x) dx = \alpha \cdot 1 + (1 - \alpha) \cdot 1 = 1$.

Because both p_1 and p_2 are valid distributions, one also knows that $\forall x \in \Omega : \hat{p}_1(x) \geq 0$ and $\forall x \in \Omega : \hat{p}_2(x) \geq 0$, which implies that $\forall x \in \Omega : \hat{p}(x) = \alpha \hat{p}_1(x) + (1 - \alpha) \hat{p}_2(x) \geq 0$ so long as $\alpha \geq 0$ and $(1 - \alpha) \geq 0$, or equivalently, $0 \leq \alpha \leq 1$ as stipulated. \square

A.3 Truncated Stationary Distribution Lemma

Truncated Stationary Distribution Lemma. *Let Ω_1 and Ω_2 be two disjoint state spaces, and let K is a transition kernel defined on $\Omega = \Omega_1 \cup \Omega_2$. Let p be a stationary distribution of K , and let \hat{p} be the distribution p truncated to Ω_1 and renormalized. If K is such that $\forall x \in \Omega_1, x^* \in \Omega_2 : K(x^*|x) = K(x|x^*) = 0$ then \hat{p} is also a stationary distribution of K .*

Proof. \hat{p} is a renormalized truncation of p to Ω_1 , that is:

$$\hat{p}(x) = \begin{cases} \frac{1}{z} p(x), & \text{if } x \in \Omega_1; \\ 0, & \text{if } x \in \Omega_2; \end{cases} \quad (\text{A.7})$$

where $z = \int_{\Omega_1} p(x) dx$. \hat{p} is a stationary distribution of K exactly when:

$$\hat{p}(x^*) = \int_{\Omega} \hat{p}(x) K(x^*|x) dx \quad (\text{A.8})$$

$$= \int_{\Omega_1} \hat{p}(x) K(x^*|x) dx + \int_{\Omega_2} \hat{p}(x) K(x^*|x) dx \quad (\text{A.9})$$

Because Ω_1 and Ω_2 are disjoint, x^* is either in Ω_1 or in Ω_2 ; consider these cases separately.

Case 1: $x^* \in \Omega_1$. In this case, equation A.9 reduces to

$$\frac{1}{z} p(x^*) = \int_{\Omega_1} \frac{1}{z} p(x) K(x^*|x) dx + \int_{\Omega_2} \hat{p}(x) K(x^*|x) dx \quad (\text{A.10})$$

Because $K(x^*|x) = 0$ for $x \in \Omega_2, x^* \in \Omega_1$, it is the case that $\int_{\Omega_2} \hat{p}(x)K(x^*|x)dx = 0 = \int_{\Omega_2} \frac{1}{z}p(x)K(x^*|x)dx$, and therefore:

$$\frac{1}{z}p(x^*) = \int_{\Omega_1} \frac{1}{z}p(x)K(x^*|x)dx + \int_{\Omega_2} \frac{1}{z}p(x)K(x^*|x)dx \quad (\text{A.11})$$

Canceling the $\frac{1}{z}$ factors and combining the integrals produces:

$$p(x^*) = \int_{\Omega} p(x)K(x^*|x)dx \quad (\text{A.12})$$

This must hold because p was stipulated to be a stationary distribution of K .

Case 2: $x^ \in \Omega_2$.* In this case, equation A.9 reduces to

$$0 = \int_{\Omega_1} \frac{1}{z}p(x)0 dx + \int_{\Omega_2} 0 K(x^*|x)dx \quad (\text{A.13})$$

which is clearly true. □

Appendix B

BLAISE SDK Legend

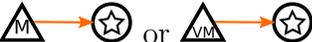
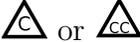
States

Symbol	Description	Page
	State	39
	Root State	40
	Collection State	41
	State→State edge	39
	State→State dependency edge	43

Densities

Symbol	Description	Page
	Density	47
	Root Density	45
	Multiplicative Density	48
	Multiplicative Collection Density	49
	Associated Collection Density	66
	Density→Density edge	47
	Density→State edge	47

Kernels

Symbol	Description	Page
	Kernel	49
	Root Kernel	51
	Initialization Kernel	76
	Concrete Mixture Kernel	56
	Virtual Mixture Kernel	66
	Concrete Cycle Kernel	55
	Virtual Cycle Kernel	66
	Metropolis-Hastings Kernel	58
	Gibbs Kernel	58
	Conditional Hybrid Kernel	56
	Let Kernel	118
	Kernel→Kernel edge	55
	Kernel→State edge	50

Other

Symbol	Description	Page
	Structure repetition highlight	41

Bibliography

- [1] Wikipedia, the free encyclopedia. <http://www.wikipedia.org>.
- [2] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, second edition, 1996.
- [3] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.
- [4] M. J. Beal, Z. Ghahramani, and C. E. Rasmussen. The infinite hidden Markov model. In *Advances in Neural Information Processing Systems 14*. MIT Press, 2002.
- [5] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3, March 2003.
- [6] David M. Blei, Thomas L. Griffiths, Michael I. Jordan, and Joshua B. Tenenbaum. Hierarchical topic models and the nested chinese restaurant process. In *Advances in Neural Information Processing Systems*, 2004.
- [7] Steve Burbeck. Applications programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC). <http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>, 1987.
- [8] Peter Carbonetto, Jacek Kisynski, Nando de Freitas, and David Poole. Non-parametric bayesian logic. In *Proceedings of the 21th Annual Conference on Uncertainty in Artificial Intelligence (UAI-05)*, Arlington, Virginia, 2005. AUAI Press.
- [9] Nick Chater, Joshua B. Tenenbaum, and Alan Yuille. Probabilistic models of cognition: Conceptual foundations. *Trends in Cognitive Sciences*, 10(7):287–291, July 2006.
- [10] Allan M. Collins and Ross M. Quillian. Retrieval time from semantic memory. *Journal of verbal learning and verbal behavior*, 8:240–248, 1969.
- [11] G. Cooper and S. Krishnamurthi. FrTime: Functional reactive programming in PLT Scheme. <http://citeseer.ist.psu.edu/cooper04frtime.html>, 2004.

- [12] Beau Cronin, Mriganka Sur, and Konrad Koerding. Hierarchical Bayesian modeling and Markov chain Monte Carlo sampling for tuning curve analysis. *Journal of Neuroscience*, 2007. In Preparation.
- [13] W. C. Dash. The growth of silicon crystals free from dislocations. *Proc. Intl. Conf. on Crystal Growth*, 29(4):361385, 1958.
- [14] Arnaud Doucet, Nando Defreitas, and Neil Gordon. *Sequential Monte Carlo Methods in Practice*. Springer, June 2001.
- [15] David J. Earl and Michael W. Deem. Parallel tempering: Theory, applications, and new perspectives. *Physical Chemistry Chemical Physics*, 7:3910–3916, 2005.
- [16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.
- [17] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, November 1984.
- [18] C. Geyer. Markov chain Monte Carlo maximum likelihood. In *Computing Science and Statistics: Proceedings of the 23rd Symposium on the Interface*, volume 2, pages 156–163, 1991.
- [19] Walter R. Gilks and Carlo Berzuini. Following a moving target—Monte Carlo inference for dynamic Bayesian models. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 63(1):127–146, 2001.
- [20] Noah D. Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua Tenenbaum. Church: a language for generative models with non-parametric memoization and approximate inference. In *Uncertainty in Artificial Intelligence*, 2008. In Review.
- [21] Peter J. Green. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82:711–732, 1995.
- [22] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, April 2004.
- [23] Thomas L. Griffiths and Joshua B. Tenenbaum. Two proposals for causal grammars. In Alison Gopnik and Laura Schulz, editors, *Causal learning: Psychology, philosophy, and computation*. Oxford University Press, 2007. In Press.
- [24] Tom Griffiths and Zoubin Ghahramani. Infinite latent feature models and the Indian buffet process. In *NIPS*, 2005.
- [25] GroupLens. MovieLens data set. <http://www.grouplens.org/node/12>, 2006.
- [26] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, April 1970.

- [27] D. Husmeier, R. Dybowski, and S. Roberts, editors. *Probabilistic modelling in bioinformatics and medical informatics*. Springer-Verlag, London, UK, 2005.
- [28] E. T. Jaynes. *Probability Theory - The Logic of Science*. Cambridge University Press, Cambridge, 2003.
- [29] Charles Kemp and Joshua B. Tenenbaum. Theory-based induction. In *Proceedings of the 25th Annual Conference of the Cognitive Science Society*, 2003.
- [30] Charles Kemp, Joshua B. Tenenbaum, Thomas L. Griffiths, Takeshi Yamada, and Naonori Ueda. Learning systems of concepts with an infinite relational model. In *AAAI*. AAAI Press, 2006.
- [31] J. H. Kim and J. Pearl. A computational model for combined causal and diagnostic reasoning in inference systems. In *Proceedings of the IJCAI-83*, pages 190–193, Karlsruhe, Germany, 1983.
- [32] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220, 4598(4598):671–680, May 1983.
- [33] K. Korb and A. Nicholson. *Bayesian artificial intelligence*. Chapman and Hall, Boca Raton, Florida, 2004.
- [34] G. Krasner and S. Pope. A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.
- [35] Mark Loy, Robert Eckstein, Dave Wood, James Elliott, and Brian Cole. *Java Swing, Second Edition*. O’Reilly, Sebastopol, CA, November 2002.
- [36] David J. Lunn, Nicky Best, and John Whittaker. Generic reversible jump MCMC using graphical models. Technical Report EPH-2005-01, Imperial College London, 2005.
- [37] Wei J. Ma, Jeffrey M. Beck, Peter E. Latham, and Alexandre Pouget. Bayesian inference with probabilistic population codes. *Nature Neuroscience*, 9(11):1432–1438, October 2006.
- [38] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003. available from <http://www.inference.phy.cam.ac.uk/mackay/itila/>.
- [39] Vikash Mansinghka, Keith Bonawitz, Eric Jonas, and Josh Tenenbaum. Efficient Monte Carlo inference for infinite relational models. Neural Information Processing Systems: presented at workshop on statistical models of networks, 2007. Available electronically at <http://www.cs.ubc.ca/~murphyk/nips07NetworkWorkshop/>.

- [40] Vikash Mansinghka, Daniel Roy, Keith Bonawitz, Eric Jonas, and Joshua Tenenbaum. Approximate Monte Carlo inference via systematic stochastic search. In *Uncertainty in Artificial Intelligence*, 2008. In Review.
- [41] R. Mao, J. Schummers, D. T. Page, C. Lee, C. Kim, J. L. R. Rubenstein, and M. Sur. Effects of subtype-specific loss of inhibitory interneurons on stimulus-specific responses in visual cortex. Poster 920.24/NN9, Society for Neuroscience, 2007.
- [42] A. T. Mccray. An upper-level ontology for the biomedical domain. *Comparative and Functional Genomics*, 4, January/February 2003.
- [43] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machine. *J. Chem. Phys.*, 21:1087–1091, 1953.
- [44] Brian Milch, Bhaskara Marthi, David Sontag, Stuart Russell, Daniel L. Ong, and Andrey Kolobov. Approximate inference for infinite contingent bayesian networks. In Robert G. Cowell and Zoubin Ghahramani, editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, Jan 6-8, 2005, Savannah Hotel, Barbados*, pages 238–245. Society for Artificial Intelligence and Statistics, 2005. (Available electronically at <http://www.gatsby.ucl.ac.uk/aistats/>).
- [45] Brian Christopher Milch. *Probabilistic Models with Unknown Objects*. PhD thesis, EECS Department, University of California, Berkeley, December 13 2006.
- [46] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo samplers. *Journal Of The Royal Statistical Society Series B*, 68(3):411–436, 2006.
- [47] Kevin P. Murphy. The Bayes Net Toolbox for Matlab. *Computing Science and Statistics*, 2001.
- [48] Richard E. Neapolitan and Xia Jiang. *Probabilistic Methods for Financial and Marketing Informatics*. Morgan Kaufmann, San Mateo, CA, 2007.
- [49] J. Pearl. Reverend bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the American Association of Artificial Intelligence National Conference on AI*, pages 133–136, Pittsburgh, PA, 1982.
- [50] Avi Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI*, pages 733–740, 2001.
- [51] Avi Pfeffer. The design and implementation of IBAL: A general-purpose programming language. In Lise Getoor and Benjamin Taskar, editors, *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

- [52] A. Pouget, P. Dayan, and R. S. Zemel. Inference and computation with population codes. *Annu Rev Neurosci*, 26:381–410, 2003.
- [53] Gian-Carlo Rota. The number of partitions of a set. *American Mathematical Monthly*, 71, 1964.
- [54] Daniel Roy, Charles Kemp, Vikash Mansinghka, and Joshua Tenenbaum. Learning annotated hierarchies from relational data. In *Advances in Neural Information Processing Systems*, volume 19, 2007.
- [55] N. C. Rust, O. Schwartz, J. A. Movshon, and E. P. Simoncelli. Spatiotemporal elements of macaque v1 receptive fields. *Neuron*, 46(6):945–956, June 2005.
- [56] Maneesh Sahani and Jennifer F. Linden. Evidence optimization techniques for estimating stimulus-response functions. In *Advances in Neural Information Processing Systems*, volume 15, pages 301–308, 2003.
- [57] James Schummers, Beau Cronin, Klaus Wimmer, Marcel Stimberg, Robert Martin, Klaus Obermayer, Konrad Koerding, and Mriganka Sur. Dynamics of orientation in cat V1 neurons depend on location within layers and orientation maps. *Frontiers in Neuroscience*, 1(1):145–159, 2007.
- [58] David Spiegelhalter, Andrew Thomas, Nicky Best, and Dave Lunn. *WinBUGS: User Manual, Version 2.10*. Medical Research Council Biostatistics Unit, 2005.
- [59] Bruce A. Tate and Curt Hibbs. *Ruby on Rails: Up and Running*. O’Reilly, Sebastopol, CA, August 2006.
- [60] Joshua B. Tenenbaum, Thomas L. Griffiths, and Charles Kemp. Theory-based bayesian models of inductive learning and reasoning. *Trends in Cognitive Sciences*, 10(7):309–318, July 2006.
- [61] Joshua B. Tenenbaum and Fei Xu. Word learning as Bayesian inference. In *Proceedings of the 22nd Annual Conference of the Cognitive Science Society*, 2000.
- [62] Andrew Thomas, Bob O’Hara, Uwe Ligges, and Sibylle Sturtz. Making BUGS open. *R News*, 6/1:12–17, March 2006.
- [63] Trolltech. *Qt Reference Documentation*.
- [64] Zhuowen Tu, Song-Chun Zhu, and Heung-Yeung Shum. Image segmentation by data driven Markov chain Monte Carlo. In *Proceedings of the Eighth IEEE International Conference on Computer Vision*, volume 2, pages 131–138, 2001.
- [65] Shimon Ullman. *High Level Vision*, chapter 10, pages 317–358. Sequence Seeking and Counter Streams: A Model for Information Flow in the Visual Cortex. MIT Press, Cambridge, Massachusetts, 1996.

- [66] J. M. Winn. *Variational Message Passing and its Applications*. PhD thesis, University of Cambridge, October 2003.
- [67] Alan Yuille and Daniel Kersten. Vision as Bayesian inference: analysis by synthesis? *Trends in Cognitive Sciences*, 10(7):301–308, July 2006.