Modeling Cognition with Probabilistic Programs: Representations and Algorithms

by

Andreas Stuhlmüller

Submitted to the Department of Brain and Cognitive Sciences
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Cognitive Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

Author . . . .
Department of Brain and Cognitive Sciences
March 28, 2015

Certified by . . . .
Noah D. Goodman
Assistant Professor, Stanford University
Thesis Supervisor

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

Accepted by . . . .
Matthew A. Wilson
Sherman Fairchild Professor of Neuroscience and Picower Scholar
Director of Graduate Education for Brain and Cognitive Sciences

Modeling Cognition with Probabilistic Programs: Representations and Algorithms

by

Andreas Stuhlmüller

Submitted to the Department of Brain and Cognitive Sciences
on March 28, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Cognitive Science

Abstract

This thesis develops probabilistic programming as a productive metaphor for understanding cognition, both with respect to mental representations and the manipulation of such representations.

In the first half of the thesis, I demonstrate the representational power of probabilistic programs in the domains of concept learning and social reasoning. I provide examples of richly structured concepts, defined in terms of systems of relations, subparts, and recursive embeddings, that are naturally expressed as programs and show initial experimental evidence that they match human generalization patterns. I then proceed to models of reasoning about reasoning, a domain where the expressive power of probabilistic programs is necessary to formalize our intuitive domain understanding due to the fact that, unlike previous formalisms, probabilistic programs allow conditioning to be *represented in* a model, not just *applied to* a model. I illustrate this insight with programs that model nested reasoning in game theory, artificial intelligence, and linguistics.

In the second half, I develop three inference algorithms with the dual intent of showing how to efficiently compute the marginal distributions defined by probabilistic programs, and providing building blocks for process-level accounts of human cognition. First, I describe a Dynamic Programming algorithm for computing the marginal distribution of discrete probabilistic programs by compiling to systems of equations and show that it can make inference in models of "reasoning about reasoning" tractable by merging and reusing subcomputations. Second, I introduce the setting of amortized inference and show how learning inverse models lets us leverage

samples generated by other inference algorithms to compile probabilistic models into fast recognition functions. Third, I develop a generic approach to coarse-to-fine inference in probabilistic programs and provide evidence that it can speed up inference in models with large state spaces that have appropriate hierarchical structure.

Finally, I substantiate the claim that probabilistic programming is a *productive* metaphor by outlining new research questions that have been opened up by this line of investigation.

Thesis Supervisor: Noah D. Goodman
Title: Assistant Professor, Stanford University

Thesis Supervisor: Joshua B. Tenenbaum
Title: Paul E. Newton Career Development Professor

Acknowledgments

I had two caring mentors, and many others who made this thesis what it is.

First and foremost, I thank Noah Goodman—for teaching me most of what I know about cognitive science, for constraining my random walk through project space just the right amount, for being sufficiently ambitious, and for fresh ideas, even after all these years.

Josh Tenenbaum, for responding to my out-of-the-blue e-mail many years ago (which set this entire project in motion), for his good taste in research, for much-needed guidance early on, and for letting me explore, even when those explorations led me to the other side of the country.

But let's start at the beginning, which for our purposes is Osnabrück.

I thank Elmar Cohors-Fresenborg for the class "Formalisierung von Wissen", which shaped my thinking about the structure of knowledge; and Frauke Harms, Paula Henk, Falk Lieder, and Jonas Lorenz, for intellectual companionship when we encountered these ideas together.

Lena Kästner, Katharina Wilmes, Corinna Zennig, Tim Schröder, Nico Möller, Benedict Küster, Nicole Troxler, Lucas Theis, Alexandra Surdina, and Christiane Mietzsch, for making college feel social and warm; and Tobias Grage, for Rosenstolz.

Natalie Schaworonkow, for unfailing authenticity.

Frauke Harms, for lasting friendship.

Sven Walter, Achim Stephan, and Kai-Uwe Kühnberger, without whom my stay at MIT wouldn't have been possible; and Frank Jäkel, without whom I would have felt alone when I first got there.

At MIT, I met some who are now my closest friends.

I thank Owain Evans, for charitable and reasoned engagement with weird ideas.

Leon Bergen, for equally reasoned engagement with not-so-weird ideas.

John McCoy, for friendship, inventiveness, and for letting me live in his basement.

Jon Malmaud, for saying things you can't say.

Tomer Ullman, for his axiomatic and well-defined friendship, and for demonstrating that being humorous doesn't equal being sarcastic.

Tim O'Donnell, for friendship and scholarship.

Ted Gibson and Leslie Kaelbling, for contributing their experience in computational linguistics and hierarchical planning (respectively) as part of my thesis committee, and for having reasonable expectations about my knowledge in these domains.

I am grateful to many others at CoCoSci and around MIT, for research ideas, friendship, or (most commonly) both: David Wingate, Cameron Freer, Chris Baker, Dan Roy, Roger Grosse, Tobias Gerstenberg, Vikash Mansinghka, Josh Hartsthorne, Sam Gershman, Eyal Dechter, Steve Piantadosi, Yarden Katz, and Peter Battaglia.

At Stanford, I have come to know another group of people without whom this PhD would not have been the same.

I thank Daniel Hawthorne, for his jovial nature.

Long Ouyang, for asking the tough questions.

My other friends and collaborators in CoCoLab and around Stanford: Erin Bennett, Irvin Hwang, Justine Kao, Desmond Ong, Siddharth N, Michael-Henry Tessler, Thomas Icard, Daniel Selsam, Judith Degen, Daniel Ritchie, Greg Scontras, Daniel Ly, William Leif Hamilton, Jacob Steinhardt, Yoni Donner, and Ling Yang.

Many others have taught me over the years. I thank Jürgen Schmidhuber, Alexander Förster, and Tom Schaul, for welcoming me at IDSIA when I knew nothing. Nick Bostrom and Daniel Dewey, for welcoming me at FHI when they knew nothing about me. Moshe Looks and Nick Hay, for a fun summer at Google. Vladimir Nesov, for teaching me the importance of evaluating arguments independent of their conclu-

sions. And Paul Christiano, for demonstrating what it means to take "doing good" seriously, and how to actually sit down and think.

Essentially all writing in this thesis is based on published work, which would not exist if not for my collaborators. Besides Noah Goodman, who has guided the research behind all chapters, including Chapter 2 [83], my collaborators for Chapter 3 have been Josh Tenenbaum and Irvin Hwang [39, 84], Tim O'Donnell and Dan Roy for early stage work on the ideas that led to Chapter 5 [82], Jessica Taylor for Chapter 6 [85], and Robert Hawkins and Siddharth N for Chapter 7 [86]. The introduction and conclusion make use of bits and pieces from each of these publications.

Finally, I thank Sha, for being my companion.

And my family, for everything.

Contents

1 Introduction **17**

2 Background: Probabilistic Programming **29**

I Representations **37**

3 Concept Learning as Program Induction **39**

3.1 Introduction 39

3.2 Formal framework 42

3.2.1 Concept representation 42

3.2.2 Categorization 45

3.3 Experiment 48

3.3.1 Setup 48

3.3.2 Results 52

3.4 Learning, revisited 56

3.4.1 Bayesian model merging 57

3.4.2 Bayesian program merging 58

3.5 Conclusion 60

4 Reasoning about Reasoning as Nested Conditioning 65
4.1 Introduction . . . 65
4.2 Modeling theory of mind as nested conditioning . . . 67
4.3 Schelling coordination games . . . 68
4.4 Language understanding . . . 71
4.5 Game playing . . . 75
4.6 Induction puzzles . . . 77
4.7 Discussion . . . 82
4.8 Conclusion . . . 83

II Algorithms 85

5 Dynamic Programming for Probabilistic Programs 87
5.1 Introduction . . . 87
5.2 Inference as marginalization . . . 89
5.3 Multiply-intractable distributions . . . 90
5.4 A Dynamic Programming algorithm . . . 94
5.4.1 Approach . . . 95
5.4.2 Algorithm . . . 99
5.4.3 Technical ingredients . . . 104
5.5 Empirical evaluation . . . 105
5.6 Related work . . . 112
5.7 Conclusion . . . 114

6 Learning Stochastic Inverses 117
6.1 Introduction . . . 117
6.2 Inverse factorizations . . . 120

6.3 Learning stochastic inverses . . . 121
6.4 Inverse MCMC . . . 124
6.5 Experiments . . . 127
6.6 Related work . . . 131
6.7 Conclusion . . . 134

7 Coarse-to-Fine Sequential Monte Carlo 135

7.1 Introduction . . . 135
7.2 Background . . . 138
7.2.1 Probabilistic programming in WebPPL . . . 138
7.2.2 Sequential Monte Carlo . . . 140
7.3 Algorithm . . . 141
7.3.1 Heuristic factors . . . 143
7.3.2 Prerequisites . . . 144
7.3.3 Model transform . . . 145
7.3.4 Lifting constants . . . 146
7.3.5 Lifting random variables . . . 147
7.3.6 Lifting factors . . . 148
7.3.7 Lifting primitive functions . . . 149
7.4 Empirical evaluation . . . 150
7.4.1 Markov Random Fields . . . 152
7.4.2 Factorial HMM . . . 156
7.4.3 Visual scene understanding . . . 158
7.5 Discussion . . . 159

8 The Road Ahead 161

List of Figures

2-1 A Binomial(5, .5) distribution. 31

2-2 A simple scenario that requires reasoning under uncertainty 32

2-3 The conditioning operator `query`, defined as a Church function 33

2-4 A hierarchical model of urn draws in Church 35

2-5 Predictions for the urn model in Figure 2-4 36

3-1 Observations generated by a simple structured generative process . . 41

3-2 Human, tree exemplar, and generative model responses 53

3-3 Program induction example: flower 62

3-4 Program induction example: simple recursion 62

3-5 Program induction example: vine 63

3-6 Program induction example: tree 63

4-1 A Schelling coordination game in Church 68

4-2 Schelling game behavior as a function of reasoning depth 69

4-3 Language understanding as recursively nested conditioning 71

4-4 Predictions for the language understanding model in Figure 4-3 . . . 72

4-5 Tic-tac-toe in Church 74

4-6 Predictions for the Tic-tac-toe model in Figure 4-5 75

4-7 A stochastic version of the Blue-Eyed Islanders puzzle in Church . . . 78

4-8 Predictions for the Blue-Eyed Islanders model in Figure 4-7 79

5-1 A simple recursive probabilistic program 89
5-2 A simple program with nested conditioning 91
5-3 Application of Dynamic Programming to a Church program 98
5-4 Church model for the rope-pulling game 108
5-5 Convergence of inference for the rope-pulling game 109
5-6 DP inference time as a function of nested conditioning depth 111
5-7 DP results for the Blue-Eyed Islanders puzzle 112

6-1 Brightness constancy Bayes net and inverse Bayes net 119
6-2 Schema of the Bayes net used in the first Inverses experiment 129
6-3 The effect of training on approximate posterior samples 130
6-4 Learning an inverse for the brightness constancy model 130
6-5 Error and acceptance rate as a function of training samples 131
6-6 Error and acceptance rate as a function of tasks 132
6-7 Effect of source of training samples 132
6-8 Results for UAI inference competition Bayes nets 133
6-9 Estimating inverses using logistic regression 133

7-1 Incremental coarsening reduces surprise in SMC 136
7-2 Heuristic factors 140
7-3 A coarse-to-fine model 142
7-4 Quantitative inference results for Markov Random Field models . . . 151
7-5 Coarsening the Ising model at the critical state 152
7-6 Inference results for a factorial HMM 156
7-7 Inference results for a simple scene understanding model 158

List of Tables

3.1 Concept types: prototypes, nested prototypes, parts 50
3.2 Concept types: parameterized parts, recursion 51
3.3 Human-model correlations for the concept-learning experiment 53
3.4 Generalization behavior in the concept-learning experiment 54

Chapter 1

Introduction

The world has changed since the rise of civilization some 10,000 years ago. Look around—what do you see? You may be sitting on a chair, in front of a computer, a glass of water next to it on the table, within the walls of a house, the neighbor's dog barking outside, cars driving by on asphalt roads built next to power lines supplying electricity for the local school or hospital. Almost every part of your environment has been shaped by humans; it is there because we intend for it to be there, or at least approve of its existence.

This has arguably been a change for the better—as indicated by the existence of the notion of *progress*—even if not without exceptions, and not without contention. Basic human needs such as food, shelter, health, and physical safety are provided to a degree far beyond hunter-gatherer times. What is responsible for this change? While it may be difficult to pin down the relative contributions of different causes and enabling factors, it is safe to say that our capability for *thought* was a necessary ingredient. Let me explain.

Many things can be discovered by trial and error. For example, it is easy to

imagine that you might discover by accident that nuts can be cracked by hitting them with large rocks, revealing their tasty kernel. Even complex outcomes can be reached by trial and error if there is a local feedback signal that helps select future trials. Most notably, evolution has produced organisms of astonishing complexity by accumulating incremental changes over time, each the result of blind trial and error in an environment that systematically favored some types of changes over others.

However, certain outcomes are beyond the reach of blind trial and error, or at least would require trial and error on evolutionary timescales.[1] Consider what it takes to be a successful farmer in a dry region. Over the course of many months, you water fields and gain little. It is only at the end of the season that you (quite literally) reap the harvest of your hard work. And some strategies you might use to be successful, such as crop rotation, take even longer to pay off. In general, if you need to get n successive steps right to accomplish a goal, and if you have k choices at each step and no local signal that indicates whether you are on the right track, your chance of success is $1/k^n$, a chance that rapidly shrinks as n and k grow. Not to mention the fact that trying some things might kill you! Now consider the construction of most of the artifacts and systems that form the backdrop against which we live our lives, and it is clear that they involved getting many sequential steps right. Imagine constructing a power plant by blind trial and error! How, then, did they come to exist in such a short period of time?

The first key insight is that you don't always need to try things in the real world. If you can build a simulation that captures the relevant aspects of the world, you can try things in this simulation and observe what their effects would have been, had you tried the same thing in the real world. (The simulations we are most interested

[1]Humans evolved, after all, so *in some sense* all our accomplishments are the result of the blind trial-and-error process of evolution.

in are, of course, instantiated in our minds.) This can be much quicker and much less costly than executing trial and error in the real world. Not everything matters equally, and so simulations can have many fewer components than the real thing. Furthermore, it is preferable to die or suffer in a simulation than to experience the real-world equivalent. Such simulations require a *representation* of some parts of the external world; a *mental representation.* We will also refer to such representations and their component parts as *models* and *concepts*, and to the idea of modeling only certain features of the world as *abstraction.*

However, mental simulation has its limits. As we consider outcomes that require longer plans, and plans that require us to choose between more options at each step, the probability of succeeding by blind trial and error shrinks exponentially. If a plan has 10 steps, and 10 different things you could do at each step, there are 10 billion things to try; and in the real world, the number of meaningfully distinct things one could do at each point in time is likely much larger than 10, depending on the grain of analysis.

The second key insight is that we can manipulate mental representations in ways that don't apply to the world itself. We don't need to randomly try sequences of actions until we succeed. Instead, we can—for example—reason backwards from a goal until we find a sequence of steps that starts from our current situation. More generally, we can apply conditional reasoning: suppose that some fact is true in our simulation, then what follows? If the simulation is accurate with respect to the relevant factors, and our reasoning methods are truth-preserving, our conclusions will be accurate as well. We will refer to this kind of reasoning as *inference*, and it applies in many settings, not just goal-directed reasoning. In fact, even learning what simulation best captures the world (based on what we have observed) can be seen as a kind of inference.

These two concepts—mental representation and inference—are key ingredients for our ability to shape the world, and they form the heart of this thesis. At the beginning of this introduction, we have seen some of the impacts we have had on our environment, and some of the ways in which we have made it more humane, thanks to our planning and reasoning abilities (among other contributors). However, much remains to be done. Even today, disease, war, and poverty are serious problems, and the list of problems that are somewhat less serious, but still of great importance, is too long to put into words. We would like to build machines that can help us accomplish the non-trivial tasks required to fix these problems—tasks that involve modeling the world and making choices that steer it into very particular directions that we would never encounter by blind trial and error.

There are different ways to go about this. One could start from scratch, or one could try to understand and reproduce the architecture of the human mind at various levels of fidelity. There are arguments either way, and both approaches should be pursued. In this thesis, I pursue an approach based on understanding human thought and building machines that think "like humans" in relevant ways (which I will elaborate on shortly).

I follow this approach for three reasons, two common and one less common. *First*, following the template provided by the human mind might be easier. *Second*, we would like to understand how the human mind works for independent reasons, such as improving education, curing mental illness, and augmenting human intellect, and building a mind is a great tool for understanding it. In Richard Feynman's words, "What I cannot create, I do not understand". *Third*, we might be able build more useful machines this way. This reason is usually less emphasized, but perhaps more important in the long run. Whenever we pose a task, be it to a human or a machine, we have to specify it in some form. For example, we can give a recipe for the

individual steps involved in the task, describe the desired outcome, or give examples of correct and incorrect behavior. The correct interpretation of such instructions relies on shared background knowledge. To follow a recipe for making poached eggs, you need to know what the words "saucepan", "teaspoon", and "yolk" refer to. To learn how to drive well from examples of good and bad driving, you need to generalize in ways similar to how a human would. As we build machines that act with greater autonomy and that take on more ambitious tasks, there is more opportunity for differences in background knowledge and generalization patterns to hamper reliable delegation to machines. It is at least plausible that this can be ameliorated by building machines with mental representations that mirror human knowledge, and with mechanisms for acquiring knowledge that mirror human learning.

Once we have decided to take a path based on understanding human cognition, a second decision point is the level of abstraction to use for thinking about the mind. Since Marr [54], this is commonly viewed as a choice between the computational level, the algorithmic level, and the level of implementation. A computational-level analysis of human cognition is primarily concerned with reverse-engineering the problems that human cognition solves, and the broad approaches it takes to solve them. An algorithmic-level analysis seeks to understand the processes and algorithms that are used to solve the problems defined on the computational level. Finally, an analysis on the level of implementation is concerned with the physical (and specifically neural) implementation of those algorithms. We are primarily interested in building machines that can solve the same problems that human cognition is solving, and in ways that are broadly compatible with how humans are solving them, so we will focus on the computational level and (to a lesser extent, and only in some parts of our discussion on inference methods) on the algorithmic level. Since this approach ignores many of the specific constraints that contributed to the architecture of the human brain, the

insights derived may more naturally apply to systems that are much simpler and less sophisticated than humans, and also eventually to systems that are more complex.

What are the phenomena that we would like to understand on a computational and (to some extent) on an algorithmic level? Based on our previous discussion, the human ability to mentally represent the world and to manipulate this representation are plausibly central to many of our accomplishments. Hence, the key questions that this thesis seeks to address are: First, what are these mental representations like? Second, based on our answer to this question, what are suitable algorithms for manipulating these representations? There is a long history of prior work on both of these questions. In the next few paragraphs, I will not review this work. Instead, I will outline the most productive metaphors we currently have for understanding mental representations and algorithms that operate on mental representations, and postpone a more in-depth discussion of related work to subsequent chapters.

First, and perhaps most strikingly, mental representations are *compositional*. If you can imagine an elephant and a piano, you will have little difficulty imagining an elephant playing a piano, even if you have never seen such a thing. This compositionality reminds of the compositionality of natural language: indeed, in the previous sentence, I used compositional language to argue that our conceptual representations are similarly compositional. This similarity is responsible for the metaphor of a *language of thought* [15]. As a metaphor, natural language is difficult to use, since language itself is not fully understood, and any account of language is likely intertwined with an account of thought. However, compositionality is not restricted to *natural* language. We have artificial languages that are still compositional. Two prominent candidates are the languages of logic (and mathematics more generally), and programming languages. Of these, programming languages tend to be more procedural (describing processes), whereas logical languages tend to be more declarative

(describing state).

Second, mental representations are *graded* in the sense that they can capture uncertainty. If I tell you that my flight was delayed, you may not know how much of a delay I am talking about, but you do have a sense for what is likely and unlikely: it is probably more than a few seconds, probably less than a few days, and most likely somewhere between 15 minutes and a few hours. This gradedness in our thinking has been recognized for a long time, e.g. in prototype and exemplar models of concept learning [3, 78], which we will encounter again in Chapter 3. Uncertainty is commonly formalized in the setting of probability theory, where probabilities can be viewed as degrees of belief, and everything that follows will be informed by this *Bayesian* perspective [40].

Together, these two considerations—compositionality and uncertainty—suggest that formal languages that can express probability distributions may provide a rich substrate for understanding mental representation, and algorithms that operate on expressions in such languages may help us understand how we manipulate mental representations. If we look at our best metaphors for understanding mental operations, this hunch is confirmed and clarified. We liken thinking to how digital computers perform *calculations* on representations of numbers, and we believe that, although thinking is a biological process, it "has more in common with multiplication or sorting a list of numbers than with digestion or mitosis" [50]; we have invented *logical inference* to reflect the kind of thinking that mathematicians do, and we talk about *searching* for an answer; we perform *statistical inference* as a means of automating thinking about whether certain observed differences are "significant". Jaynes' work on *probabilistic inference* is explicitly motivated by the study of common sense, and by the automation of plausible reasoning based on incomplete information [40].

Based on these considerations, this thesis explores *probabilistic programs* as a

framework for studying human thought. In the next chapter, I will explain what probabilistic programs are in more detail. For now, a probabilistic program is a program that defines a probability distribution, usually by including random choices, and that supports probabilistic inference. The central thesis of this dissertation is that **probabilistic programs are a productive metaphor for understanding how the mind works**. They are a metaphor in that the representations that the brain actually uses may not be programs in the strictest sense, but—as I will argue—they share many properties with programs. This metaphor is productive in two senses. First, it is straightforward to implement probabilistic programs and algorithms that operate on them, which allows us to compare their predictions to data gathered in human experiments, and to explore the behavior and efficiency of various algorithms. Second, it is productive in the sense that the basic framework allows us to model a broad class of phenomena that are relevant to the study of human cognition. I will provide evidence that a research program for understanding the mind based on probabilistic programs is productive in both of these senses by instantiating it with respect to representations and inference.

The first half of the thesis is devoted to **representations**.

In Chapter 3, I explore the use of probabilistic programs in understanding human conceptual representations. Many real world concepts, such as "car", "house", and "tree", are more than simply a collection of features. These objects are richly structured and defined in terms of systems of relations, subparts, and recursive embeddings. I describe an approach to concept representation and learning that attempts to capture such structured objects. This approach builds on prior probabilistic approaches, viewing concepts as generative processes [14, 41, 68], and on recent rule-based approaches, constructing concepts inductively from a language of thought [15, 27]. Concepts are modeled as probabilistic programs that describe gen-

erative processes; these programs are described in a compositional language. In an exploratory concept learning experiment, I investigate human learning from sets of tree-like objects generated by processes that vary in their abstract structure, from simple prototypes to complex recursions. I compare human categorization judgements to predictions of the true generative process as well as a variety of exemplar-based heuristics. Finally, I present a computational approach to inducing probabilistic programs from data and show that it can learn simple programs in the domain of little trees.

In Chapter 4, I specialize the framework developed in Chapter 3 to the domain of social cognition, a domain that plays a particularly big role in human everyday thinking, and that particularly benefits from the representational power of probabilistic programs. A wide range of human reasoning patterns can be explained as conditioning in probabilistic models; however, conditioning has traditionally been viewed as an operation *applied to* such models, not *represented in* such models. I describe how probabilistic programs can explicitly represent conditioning as part of a model. This enables us to describe reasoning about others' reasoning using *nested* conditioning. Much of human reasoning is about the beliefs, desires, and intentions of other people; I use probabilistic programs to formalize these inferences in a way that captures the flexibility and inherent uncertainty of reasoning about other agents. I express examples from game theory, artificial intelligence, and linguistics as recursive probabilistic programs and illustrate how this representation language makes it easy to explore new directions in each of these fields.

The second half of the thesis is devoted to **algorithms**.

Motivated by the models of reasoning about reasoning developed in Chapter 4, Chapter 5 describes a Dynamic Programming algorithm for computing the marginal distribution of discrete probabilistic programs; that is, an exact inference algorithm

that aggressively reuses subcomputations where possible. In technical terms, this algorithm takes a functional interpreter for an arbitrary probabilistic programming language and turns it into an efficient marginalizer. Because direct caching of sub-distributions is impossible in the presence of recursion, we build a graph of dependencies between sub-distributions. This *factored sum-product network* makes (potentially cyclic) dependencies between subproblems explicit, and corresponds to a system of equations for the marginal distribution. We solve these equations by fixed-point iteration in topological order. I illustrate this algorithm on examples used in teaching probabilistic models, computational cognitive science research, and game theory.

Chapter 6 tackles the following puzzle: Human recognition of words, objects, and scenes is extremely rapid, often taking only a few hundred milliseconds. By contrast, Bayesian inference is computationally expensive, and even approximate, sampling-based algorithms tend to take many iterations before they produce reasonable answers. How can we reconcile the speed of recognition with the expense of coherent probabilistic inference? The Dynamic Programming algorithm developed in Chapter 5 cannot resolve this puzzle, since many models are not amenable to exact sharing of subcomputations. Therefore, in Chapter 6, I develop a class of algorithms for *amortized inference* in Bayesian networks. In this setting, we invest computation upfront to support rapid online inference for a wide range of queries. My approach is based on learning an inverse factorization of a model's joint distribution: a factorization that turns observations into root nodes. I present algorithms that accumulate information to estimate the local conditional distributions that constitute such a factorization. These *stochastic inverses* can be used to invert each of the computation steps leading to an observation, sampling *backwards* in order to quickly find a likely explanation. I show that estimated inverses converge asymptotically in the number of (prior or posterior) training samples. To make use of inverses before convergence, I

describe the *Inverse MCMC* algorithm, which uses stochastic inverses to make block proposals for a Metropolis-Hastings sampler. I explore the efficiency of this sampler for a variety of parameter regimes and Bayes nets.

In Chapter 7, I ask what it would take to scale up the models developed in previous chapters to large state spaces. When we as humans are faced a complex reasoning task, it often helps to take a step back, try to understand the big picture, and then focus on what seems most promising. In other words, we consider a very approximate (coarse) solution to the problem first, then refine to more detailed answers. I explore this idea in the setting of Sequential Monte Carlo algorithms, a class of algorithms that is based on the idea of sampling from a sequence of distributions that interpolate between a tractable distribution and an intractable distribution of interest. Usually, the sequences used are simple, e.g., based on geometric averages between distributions. When models are expressed as probabilistic programs, the models themselves are highly structured objects that can be used to derive annealing sequences that are more sensitive to domain structure. I propose an algorithm for transforming probabilistic programs to *coarse-to-fine programs* which have the same marginal distribution as the original programs, but generate the data at increasing levels of detail, from coarse to fine. I apply this algorithm to three models in the domain of tracking partially observable objects over time given visual information and show that the use of coarse-to-fine models can make existing generic inference algorithms more efficient when abstractions match domain structure.

Finally, in Chapter 8, I review some promising next steps for the project of understanding the human mind using probabilistic programs, with an eye towards building useful machines.

Chapter 2

Background: Probabilistic Programming

A probabilistic program is a program in a universal programming language with primitives for sampling from probability distributions, such as Bernoulli, Gaussian, and Poisson. Execution of such a program leads to a series of computations and random choices. Probabilistic programs thus describe models of the stochastic generation of results, implying a distribution on return values. Most of our examples use Church [26], a probabilistic programming language based on the stochastic lambda calculus. This calculus is universal in the sense that it can be used to define any computable discrete probability distribution [49] (and indeed, continuous distributions when encoded via rational approximation).

Church is a close relative of the functional programming language Scheme [1]. In this language, function application is written in prefix notation:

```
(+ 3 2)  →  5
```

This chapter is based on Stuhlmüller and Goodman [83] and Stuhlmüller et al. [86].

The same applies to conditionals:

```
(if (> 3 2) true false)  →  true
```

Functions are first-class values and can be defined using λ. For example,

```
(λ (x) (* x 2))  →  <function object>
```

refers to a function that doubles its argument. Values can be bound to variables using `define` and, for explicit scope, with `let`:

```
(let ([y 3]) (+ y 4))  →  7
```

For function definitions,

```
(define (double x) (* x 2))
```

is short for:

```
(define double (λ (x) (* x 2)))
```

The random primitive `(flip` p`)` samples from a Bernoulli distribution: it returns `true` with probability p, `false` with probability $1 - p$. By composing random primitives such as `flip` with deterministic computation, we can build complex distributions. For example, the expression

```
(sum (repeat 5 (λ () (if (flip .5) 0 1))))
```

induces the Binomial distribution shown in Figure 2-1. The meaning of a complex Church program can be understood via its *sampling semantics*: a single execution returns a sample from the distribution defined by the program; the histogram of (many) samples from the program defines its distribution on return values.

A Church program describes knowledge with uncertainty, and can be used to capture many effects of reasoning under uncertainty. As an example of reasoning under uncertainty, consider the following situation, depicted in Figure 2-2: We are presented with three opaque urns, each of which contains some unknown number of

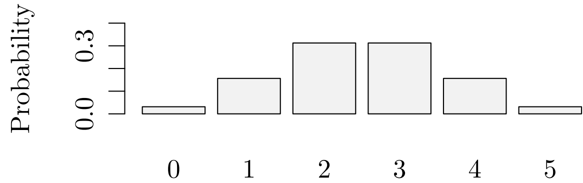


Figure 2-1: A Binomial(5, .5) distribution.

red and black balls. We do not know the proportion of red balls in each urn, and we don't know how similar the proportions of red balls are between urns, but we have reason to suspect that the urns could be similar, as they all were filled at the same factory. We are asked to predict the proportion of red balls in the third urn (1) before making any observations, (2) after observing 15 balls drawn from the first urn, 14 of which are red, and (3) after observing in addition 15 balls drawn from the second urn, only one of which is red.

Intuitively, observing the first 15 balls, almost all of which are red, has two plausible explanations: either all urns have a high proportion of red balls, or all urns have differing proportions of red balls and this particular urn happens to have a high proportion of red balls. We don't know which of these is right and therefore have not learned about the overall bias of the urns yet, but we can predict a higher proportion of red balls for urn 3. After we observe in addition 15 balls drawn from urn 2, almost all of which are black, it becomes unlikely that all urns have similar proportions, hence our predicted proportion of red balls for urn 3 goes down again and we predict that the urns are biased towards very high and very low proportions of red balls. Notice that this reasoning is non-monotonic: our degree of confidence that "urns mostly contain red balls" goes up with the first piece of evidence, but

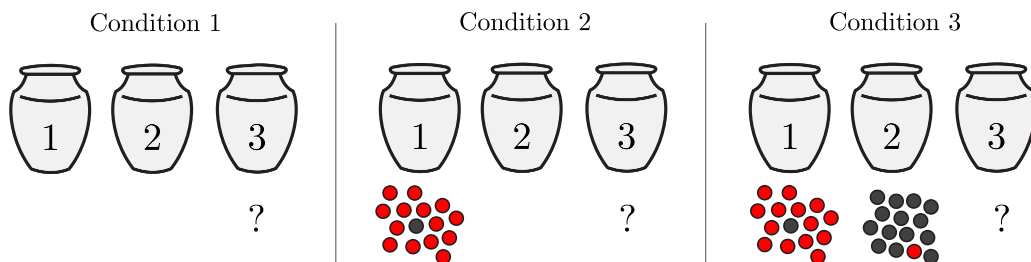


Figure 2-2: A simple scenario that requires reasoning under uncertainty: Drawing balls from urns, we want to guess for each condition (1) whether and how the urns are biased overall and (2) the likely proportion of red balls for the third urn.

decreases with the second piece.

Figure 2-4 shows how one could model this situation in Church. Figure 2-5 shows the predictions derived from the model for conditions 1-3. In Church, conditional distributions are defined using the `query` operator, which takes as arguments a list of definitions describing a generative model, a query expression, and a condition, and which returns a sample from the conditional distribution. This `query` syntax is much like the standard mathematical notation for conditionals, $P(q|c)$, but makes the model explicit. The predictions of the model in Figure 2-4 match the non-monotonic intuition sketched above. More generally, a wide range of reasoning patterns are naturally modeled as conditioning in probabilistic models, including explaining away, screening off, and Occam's razor [see 28].

In sharp contrast to less expressive modeling languages, and traditional statistical notation, conditioning is not a special primitive in Church. Figure 2-3 shows how to define a generative model that samples from a conditional distribution. The procedure `joint` draws samples from a prior distribution. The predicate `condition?` takes a sample and returns true if the condition of interest holds. The function `rejection-query` simply keeps on drawing samples from the prior until it encounters

```
(define (rejection-query joint condition?)
  (let ([sample (joint)])
    (if (condition? sample)
        sample
        (rejection-query joint condition?))))
```

Figure 2-3: The conditioning operator `query` can be defined as a Church function. It takes as arguments `joint`, a stochastic function without arguments that samples from a prior distribution, and `condition?`, a function that checks whether a given sample satisfies the condition of interest. It calls itself recursively until it finds a sample that satisfies the condition, which it then returns. The `query` syntax used in the remainder of the thesis—which takes as arguments a model, query expression, and condition—can be turned into a call to `rejection-query` using a simple syntactic transform.

a sample that satisfies the condition, in which case it returns the sample. The critical difference to simpler languages is the ability to "keep sampling until": marrying random choice with universal computation enables *stochastic recursion*, which makes it easy to define the operation of conditioning.

Of course, drawing conditional samples using `rejection-query` is very inefficient when the probability of satisfying the condition is low. Programs that contain a single level of conditioning implemented by rejection sampling already require in expectation $1/p$ recursive calls when we condition on an event with probability p. Since it is common to condition on low-probability events, this is often infeasible. For this reason, many algorithms for approximately sampling from conditional distributions have been developed, including Markov Chain Monte Carlo (MCMC), importance sampling, and variational methods.

However, we can distinguish model specification from the process that is actually used to *infer* the distribution implied by the model. From this point of view `rejection-query` is an elegant *definition* of the distribution of interest, while the

implementation we use to compute this distribution remains unspecified. Thus the practical problem of inference is the problem of efficiently computing the marginal distribution of a probabilistic program, i.e., its distribution on return values.

This separation of inference techniques and modeling assumptions is a distinguishing feature of probabilistic programming as a tool for machine learning, and it implies that any advances in algorithms provide benefits for a wide range of applications at once. Thus, while I demonstrate the inference techniques in Chapters 5-7 on small sets of models chosen for their pedagogical value, the techniques themselves apply to a much wider range of models without modification.

```
(query

  ;; model
  (define bias (uniform 0 10))
  (define red-bias (uniform 0 bias))
  (define black-bias (- bias red-bias))
  (define urn->proportion-red
    (mem
     (λ (urn)
       (beta (+ .4 red-bias) (+ .4 black-bias)))))
  (define (sample-urn urn)
    (if (flip (urn->proportion-red urn)) • •))

  ;; query expression
  (urn->proportion-red 3)

  ;; condition
  (equal? (repeat 15 (λ () (sample-urn 1)))
          (list • • • • • • • • • • • • • • •)))
```

Figure 2-4: A hierarchical model of urn draws in Church. This program formalizes reasoning about condition 2 of the urn scenario in Figure 2-2, implemented as conditional sampling using the `query` operator. It takes as arguments a model (given as a sequence of definitions), an expression of interest, and a condition (an expression that evaluates to `true` or `false`). This model first samples a bias, splits the bias into red and black, and then defines how the urns' proportions of red balls depend on the biases. The function `urn->proportion-red` is *memoized* such that it always returns the same proportion when asked about the same urn. By contrast, the function `sample-urn` always flips a new coin to determine whether to return a red or black ball, with the coin weighted by the given urn's proportion of red balls. Given this model, we sample the proportion of red balls in the third urn conditioned on observing 1 black and 14 red draws from the first urn.

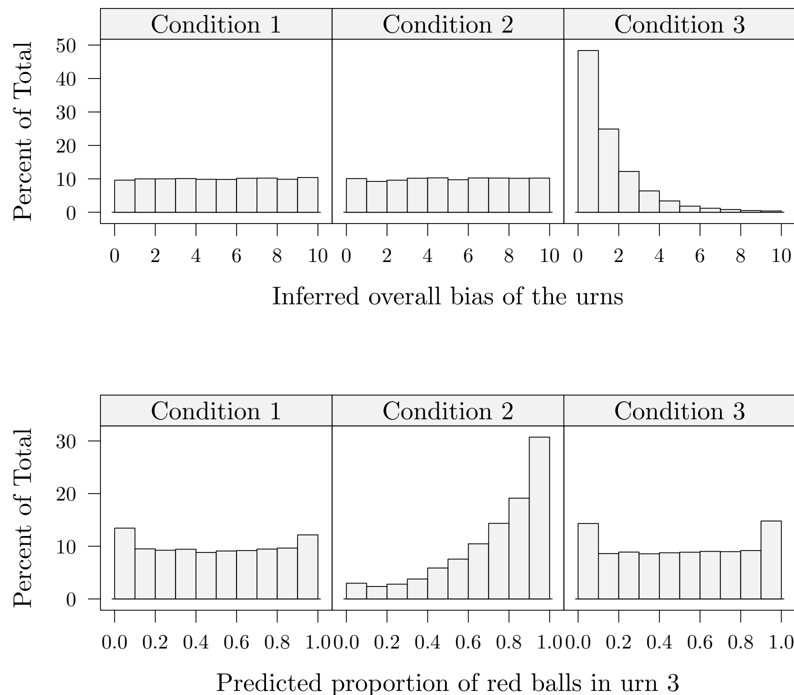


Figure 2-5: Urn scenario predictions derived from the Church model in Figure 2-4. Before seeing any draws, the model predicts uniform uncertainty about the overall bias, and mostly uniform uncertainty about the proportion of red balls in urn 3 (with extreme values being a bit more likely). After seeing 15 balls drawn from urn 1, 14 of which are red, the model still cannot judge how much the urns are biased, since it has seen only draws from a single urn. However, it would predict that, if the urns are biased, they are biased towards red, and therefore predicts that the proportion of red balls is likely to be high for urn 3. After seeing an additional 15 balls from urn 2, 14 of which are black, the model predicts that it is likely that the urns do not have directional bias, and predicts that the balls in urn 3 are most likely to be either all red or all black, but with significant probability on other proportions, since we have seen only two urns.

Part I

Representations

Chapter 3

Concept Learning as Program Induction

3.1 Introduction

Concept learning has traditionally been studied in the context of relatively unstructured objects that can be described as collections of features. Learning and categorization can be understood formally as problems of statistical inference, and a number of successful accounts of concept learning can be viewed in terms of probabilistic models defined over different ways to represent structure in feature sets, such as prototypes, exemplars, or logical rules [3, 27, 78]. Yet for many real world object concepts, such as "car", "house", "tree, or "human body", instances are more than simply a collection of features. These objects are richly structured, defined in terms of features connected in systems of relations, parts and subparts at multiple scales of abstraction, and even recursive embedding [53]. A tree has branches coming out of

This chapter is based on Stuhlmüller et al. [84] and Hwang et al. [39].

a trunk, with roots in the ground; branches give rise to smaller branches, and there are leaves at the end of the branches. A human body has a head on top of a torso; arms and legs come out of the torso, with arms ending in hands, made of fingers. A house is composed of walls, roofs, doors, and other parts arranged in characteristic functional and spatial relations that are harder to verbalize but still easy to recognize and reason about. Besides objects, examples of structured concepts can be found in language (e.g. the mutually recursive system of phrase types in a grammar), in the representation of events (e.g. a soccer match with its fixed subparts), and processes (e.g. the recipe for making a pancake with steps at different levels of abstraction).

Such concepts have not been the focus of research in the probabilistic modeling tradition. Here we describe an approach to representing structured concepts—more typical of the complexity of real world categories—using probabilistic generative processes. We test whether statistical inference with these generative processes can account for how people categorize novel instances of structured concepts and compare with more heuristic, exemplar-based approaches.

Because a structured concept like "house" has no single, simple perceptual prototype that is similar to all examples, learning such a concept might seem very difficult. However, each example of a structured concept itself has internal structure which makes it potentially very informative. Consider Figure 1, where from only a few observations of a concept it is easy to see the underlying structural regularity that can be extended to new items. The regularities underlying structured concepts can often be expressed with instructions for *generating* the examples: "Draw a sequence of brown dots, choose a branch color, and for each brown dot draw two dots of this color branching from it."

We build on the work of Goodman et al. [27], who introduced an approach to concept learning as Bayesian inference over a grammatically structured hypothesis

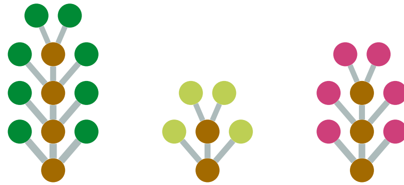


Figure 3-1: Three examples of a structured concept described by a simple generative process.

space—a "language of thought." Single concepts expressed in this language were simple propositional rules for classifying objects, but this approach naturally extends to richer representations, providing a concept learning theory for any representation language. Here we consider a language for generative processes based on *probabilistic programs*: instructions for constructing objects, which may include probabilistic choices, thus describing distributions on objects—in our case distributions on colored trees. Because this language describes generative processes as programs, it captures regularities as abstract as subparts and recursion.

The theory of concept representation that we describe here shares many aspects with previous approaches to concepts. Like prototype and mixture models [3, 30], probabilistic programs describe distributions on observations. However, prototypes and mixtures generate observations as noisy copies of ideal prototypes for the concept and thus cannot capture more abstract structures such as recursion. Like rule-based models of concept learning, our approach supports compositionality: complex concepts are composed out of simple ones—but rather than deterministic rules, our concepts denote distributions. Finally, the probabilistic program approach can be seen as a generalization of previous approaches to generative representations of concepts [14, 41, 68].

We investigate human learning for classes of generating processes that vary in their abstract structure, from simple prototypes to complex multiply recursive programs. We compare predictions for categorization judgments based on the true generative model to the predictions of exemplar models, which exploit the relational structure of the examples to varying degrees but cannot detect more abstract structure. We find two regimes: for concepts with simple prototype-like structures, human judgments are well described by a relational exemplar model, but humans can also easily learn more abstract regularities—such as sub-concepts and recursion—which are better captured by a model using more expressive generative descriptions based on probabilistic programs.

3.2 Formal framework

In the following, we first explain the formal language we use to describe generative processes, then the different methods of categorization (or generalization) we compare to subjects' judgments.

3.2.1 Concept representation

We analyze concepts as generative models, i.e. as formal descriptions of processes that generate observations. We do so within a simple domain where we can fully know and manipulate the actual generating processes behind complex objects. We use tree-structured graphs with colored nodes as observations in our experiments—these are a simple proxy for many real-world concepts, where the dependencies among parts are hierarchical or tree-like. Human bodies, buildings, and events all consist of parts that themselves contain parts, with each part standing in interesting relation

to the others.

We represent these trees as nested lists: each list denotes a tree, with the first element in the list specifying the color of the root node and the remaining elements describing the children of this node, each child itself being a list (tree). For example, the second tree shown in Figure 1 can be represented as `'(● (●) (● (●) (●)) (●))`.

We formalize the processes that generate these observations using a subset of Church, a Lisp-like stochastic programming language[1] [26]. Programs in Church describe processes that produce values; running a program corresponds to generating a value from such a process. Because Church contains primitive functions that randomly choose from a distribution on values (e.g. the function `flip` that randomly chooses `true` or `false`), Church programs describe *stochastic* processes. The meaning of a Church program is a distribution on return values—which may be complex values such as nested lists—and any given execution results in a sample from this distribution. In what follows we describe Church programs which sample colored trees.

We group generative models into classes by the abstract constructions they use. Table 3.1 illustrates each of these types using a single concept program and observations drawn from this program. The simplest tree-generating processes in our language use only the stochastic function `node`, which takes as its first argument a color symbol and as its remaining arguments subtrees. With high probability, `node` returns a tree that has the given color symbol at its root and the given subtrees as its children, but with some probability ϵ, it switches to a noise process that can return any tree, that is, `node` introduces a random noise process into the tree construc-

[1] Church uses prefix notation, i.e. function application is written with the operator first, the operands following. For example, `(node x y)` means that the function `node` is called with the arguments `x` and `y`.

tion. Under the noise process, the number of children for a node is sampled from a geometric distribution with parameter ϵ and the node color is sampled uniformly.

Programs like `(node • (node •) (node •))` denote stochastic *prototypes*. They are most likely to generate the tree that corresponds to the given colors, in this case `'(• (•) (•))`, but they can return any tree with a certain probability. The more a tree deviates from the prototype, the less likely this process is to generate it. For example, the simple program described above could switch at the third `node` to the noise process and produce `'(• (•) (• (•)))` instead of the prototype. By introducing the noise process, `node` turns a deterministic prototype into a stochastic process.

All of the more abstract ways of formalizing generative models in our tree domain compose these basic processes. *Nested prototypes* formalize the intuition that a concept or a part of a concept can be "either this or that". Running the program `(if (flip .5) (node •) (node •))` will flip a fair coin and return a sample from `(node •)` with probability .5, otherwise a sample from `(node •)`.

One of the central reasons for analyzing concepts as represented in a language of thought is that they compose analogously to the components of natural and artificial languages—*parts* similarly allow composition through reuse in our domain. A part concept is defined first and can then be used in arbitrarily many places within other concepts. For example, the program `(define (part) (node • (node •)))` names a simple part consisting of only two nodes. This part can now be reused in other concepts. For example, the most likely return value for `(node • (part) (part))` is `'(• (• (•)) (• (•)))`. When parts are defined, they are available to the noise process. This leads to some invariance to the position of parts and captures the idea that a generating process may give rise to observations that contain a part in a different place, although with lower probability compared to an observation with the part in the correct place.

Parameterized parts can capture both deterministic structure and random choices and reuse them in multiple places. When a part like `(define (part x) (node ● x x))` is used, for example in the program `(part (node ●))`, it evaluates the body of the part—here `(node ● x x)`—with `x` assigned to its argument, here `(node ●)`. Evaluating the program `(part (node ●))` is therefore most likely to result in the observation `'(● (●) (●))`.

Allowing parts to call themselves introduces *recursion*, a means to capture a large amount of repetitive observed structure in a single short definition. For example, the part `(define (p) (if (flip) (node ●) (node ● (p))))` can generate arbitrarily deep lists of single blue nodes, with shorter ones being more likely. The power of these program constructs is that they can be used compositionally to build more complex concepts, such as those shown in Table 3.1 and 3.2.

3.2.2 Categorization

In order to model generalization and categorization behavior of human subjects, we need not only a way to represent concepts, but also a way to compute the probability of any given observation belonging to a known concept. We analyze our experimental results using four models that differ in how much they make use of representational structure.

On the unstructured end of the scale, we use a model that computes generalization judgments solely by comparing the fraction of nodes that have a given color. On the other end of the scale, a generative Bayesian model uses the likelihood under the true generative process to judge category membership. In between, an exemplar model makes use of tree structure in the observations, but not of the more abstract generative process that led to the observations.

Generative Model

In modeling concept learning as Bayesian program induction, we follow the approach taken by Goodman et al. [27]. Since we formalize concepts as probabilistic programs, the likelihood $P(O|C)$ of an observation O under a given concept C corresponds to the probability of the program making its random choices such that it returns the observation as its value (see Goodman et al. [26]). The posterior probability of a concept C given observations O is proportional to this likelihood multiplied by the prior:

$$P(C|O) \propto P(O|C)P(C) \tag{3.1}$$

In the last section, we described a language for programs which generate trees; a prior $P(C)$ could be derived from this language, as in Goodman et al. [27]. An ideal learner would then infer the posterior distribution $P(\mathbf{C}|O)$ over concepts $\mathbf{C}$ given the observation O and make predictions about whether a new observation t belongs to the category of the observed objects using each concept $C \in \mathbf{C}$ in proportion to its posterior probability:

$$P(t|O) \propto \sum_{C} P(t|C)P(C|O) \tag{3.2}$$

In order to make computational modeling tractable, we make the simplifying assumptions that (1) subjects' reasoning is dominated by the maximum a posteriori (MAP) estimate of this distribution, i.e. by the single concept that has the highest posterior probability and that (2) the true generating concept C_{true} is a good approximation to the MAP estimate. Thus, for each of the concept types we investigate, we model subjects' behavior using the program from which the training data was sampled. The likelihood of a new observation t belonging to this concept is simply

$P(t|C_{true})$ which we estimate using an adaptive importance sampling algorithm.

We do not claim that subjects necessarily identify the true generating concept from a few examples; this approximation is made for computational tractability. The full Bayesian model, which maintains uncertainty over generating concepts, can make different predictions in certain cases, but it is not clear whether this represents a bias for or against the approximation—to the extent that people remain uncertain of the concept after a few examples, the Bayesian model would capture human inferences better than our approximation.

Tree Exemplar Model

This and the next two models are versions of the exemplar-based generalized context model (GCM) [62]. For observations $O_1, \ldots, O_n$ from category C and a new observation t for which we would like to estimate the likelihood under category C, we use $P(t \in C|O_1, \ldots, O_n \in C) \propto \frac{1}{n}\sum_{i=1}^{n} e^{-d(O_i,t)}$ where d is a distance measure that is sensitive to the tree structure of the observations. Starting from the root node, this measure matches the trees as much as possible, incrementing by 1 for each node that differs in color between the two trees and for each node that must be generated because it exists in one tree but not in the other tree. This approach is similar to the structure mapping approach used by Tomlinson and Love [93].

Frequency-based Exemplar Models

As in the tree exemplar model, we use a distance measure d to estimate the likelihood of an observation belonging to a category for which we have only positive examples. In this version of the model, $d(t_1, t_2)$ is the RMSE between the transition count vectors of t_1 and t_2. For each pair of node colors, the transition count vector contains

the number of times this pair occurs adjacent (as parent-child) in the given tree. We call this model *Transition GCM*. We also investigate a simplified version that uses the distance between the color count vectors. The length of this vector corresponds to the number of possible node colors, with each entry in the vector denoting how often this node color appears in the tree of interest. We call this *Set GCM*.

3.3 Experiment

This experiment is an exploratory investigation into generalization from observations of structured objects. Since our main goal in this study is to investigate the representation of concepts and their use for categorization and generalization rather than the memory aspects of learning, we use a paradigm that minimizes memory demands. By doing so, we hope to focus on how people represent the commonalities between observed instances of a concept and how they use this knowledge to generalize to new instances. We chose a domain that both contains observations with simple structure and allows for interesting generative processes—the domain of colored trees generated by probabilistic programs.

3.3.1 Setup

Participants

250 members of Amazon's crowdsourcing service *Mechanical Turk* took part in the online experiment. Subjects were compensated for participation.

Stimuli

Subjects were told that they are looking at newly discovered kinds of plants that grow in extreme environments. Each subject saw 18 pages, with each page consisting of 15 training examples, a control question, and a test example together with a classification question. Both training and test examples were images of simple trees with colored nodes drawn from tree-generating programs (see e.g. Table 3.4). For each of the concept types shown in Table 3.1, there were three tree-generating programs, and for each program there were seven test examples. These test examples were chosen to cover a wide range of both intuitive and model judgments of category membership. Both training example order and stimuli colors were randomized.

Procedure

In order to ensure that subjects process the training stimuli, a control question on each page asked how many of the training trees consist of more than 7 dots. 55 subjects answered less than 13 out of the 18 control questions correctly within an error margin of 2. We did not include these subjects in the analysis.

The categorization question asked: "How likely is it that the following plant is the same kind of plant as the plants above?" Subjects chose on a seven-step scale ranging from "certainly the same kind" to "certainly not the same kind". For each subject, the responses were normalized to $[0, 1]$.

Prototype	Nested Prototype	Parts

Prototype

```
(node •
  (node •
    (node •
      (node •)
      (node •))))
```

Nested Prototype

```
(node •
  (node •
    (node •
      (if (flip .5)
        (node •
          (node •)
          (node •
            (node •
              (node •))
            (node •)))
        (node •
          (node •)
          (node •
            (node •))
          (node •))))))
```

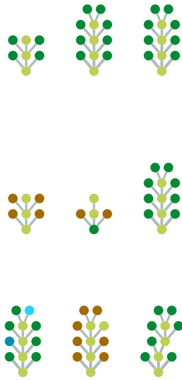
Parts

```
(define (part)
  (node •
    (node •
      (node •))))

(node •
  (part)
  (node •
    (node •
      (part))
    (part)))
```

Table 3.1: This table illustrates the concept types that can be represented within our language for generative models. For each type, an example of a concept (a stochastic program) is shown together with observations drawn from this program. The stochastic function `node` generates a mixture of the subtrees that are passed to it as its arguments and a noise process that, with low probability, can generate any tree. The abstraction methods stochastic branching, (parameterized) parts and recursion compose these stochastic prototypes into more structured generative processes.

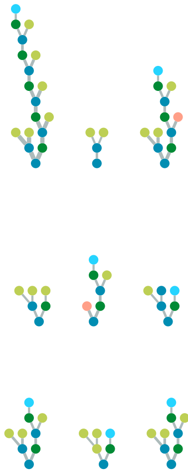
Parameterized Parts



```
(define (part x)
  (node ●
    x
    (node ●
      x
      (node ●
        x
        (node ● x x)
        x)
      x)
    x))

(part
  (if (flip .5)
    (node ●)
    (node ●)))
```

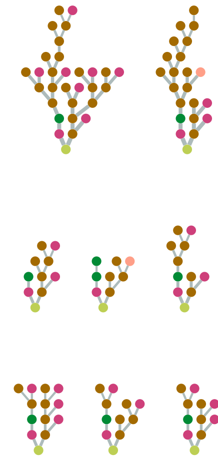
Single Recursion



```
(define (part)
  (node ●
    (if (flip .5)
      (node ●
        (part)
        (node ●))
      (node ●))))

(node ●
  (node ●
    (node ●)
    (node ●))
  (part))
```

Multiple Recursion



```
(define (part)
  (node ●
    (if (flip .3)
      (part)
      (node ●))
    (if (flip .3)
      (part)
      (node ●))))

(node ●
  (node ●
    (node ●
      (part)))
  (part))
```

Table 3.2: Table 3.1, continued.

3.3.2 Results

Table 3.3 summarizes the correlation results for all models. Figure 3-2 shows for each concept type human results and model results for both the exemplar and generative model. For each concept type, three different concepts were part of the experiment, and for each concept, seven different test observations were shown. A single point in the scatterplot contains information on the mean subject response for a single test tree and the model prediction for this tree.

Neither of the two exemplar models based on simple statistics was the best predictor for any of the concept types, with the transition-based exemplar model performing strictly better than the set-based model. An effect that is not accounted for by the less structural exemplar models is illustrated by the nested prototype example in Table 3.4: Subjects generalize significantly more to examples with branches they have seen before than to examples that have a mixture of two known branches. Likewise, subjects seem to generalize significantly more to trees with known branches than to trees that have new branches with similar surface statistics. Both results are expected under the two models that make use of tree structure.

If we group prototype and nested prototype as "less structured" and subconcepts with and without arguments, single recursions, and multiple recursions as "more structured", then the tree exemplar model best predicts human responses for the less structured stimuli whereas the true generative model best predicts performance for the more structured stimuli.

Our generative model makes the simplifying assumption that the learner infers a single generating concept from the examples whereas one interpretation of the tree exemplar model is that it uses each of the training examples as a hypothesis on what the true concept looks like. A fully Bayesian learner, which maintains a

	Set GCM	Transition GCM	Tree GCM	Generative Model
Prototype	0.589	0.751	**0.803**	0.748
Nested Prototype	0.544	0.851	**0.937**	0.904
Parts*	0.320	0.617	0.705	**0.835**
Parameterized Parts	0.298	0.591	0.778	**0.911**
Single Recursion	0.284	0.499	0.637	**0.773**
Multiple Recursion	0.505	0.561	0.451	**0.770**

Table 3.3: Human-model correlations for the experiment. Each row shows how well the different models predicted subjects' performance for a particular concept type. *Correlations excluding isolated part cases (see text).

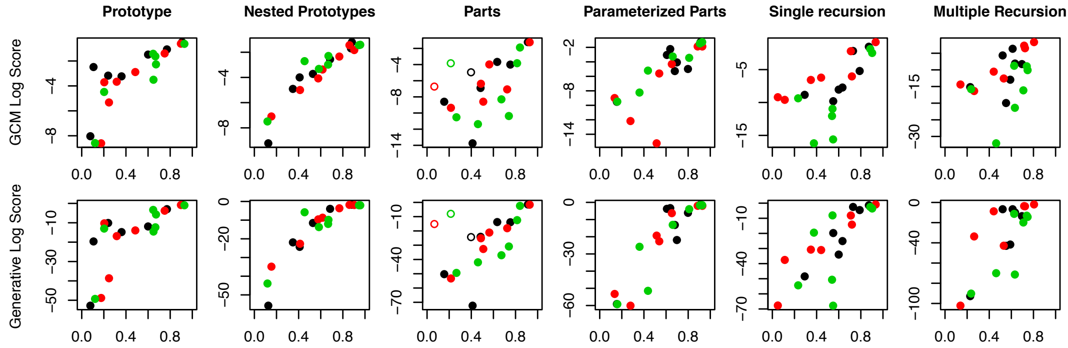


Figure 3-2: Comparison between human and model responses across concept types for tree exemplar and generative model. For each of the six concept types, three examples were shown; the color of the dots indicates to which example any given datapoint belongs. Empty circles denote isolated part cases that were excluded from the correlation analysis.

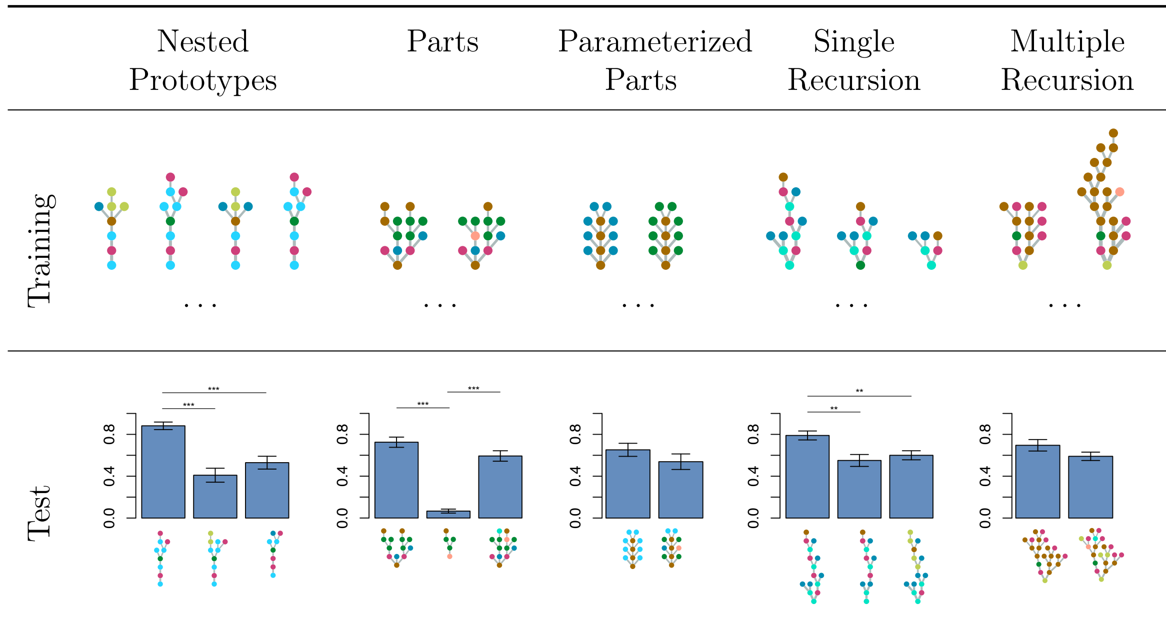


Table 3.4: This table illustrates a small selection of our experimental results. For five different concept types, training observations from a single concept of this type are shown together with subjects' generalizations for particularly interesting test examples. The error bars are standard errors of the mean.

distribution over generative processes, may predict human behavior in ways similar to the tree exemplar model for less structured examples and similar to the true generating process model for the more structured examples.

Having seen how different models predict human judgments for different concept types, we will now look at individual response patterns in order to determine ways in which both of the two structural models can be improved.

The part example in Table 3.4 shows how changes to the location of a part can have significantly different effects depending on whether the overall concept is preserved (resulting in high generalization) or whether the part is moved into a completely different environment (resulting in low generalization). By analogy, a Picasso face, with eyes in odd places, is still more of a face than an eye alone. Parts seen out of context constitute a problem for all models (except for the simplest set-based one): subjects judged these isolated parts as unlikely to come from the concept that included them as subparts whereas the models did give a high score to these examples. Since including these outliers dramatically changed the scores and made the interpretation of the model comparison difficult, we excluded these datapoints from the analysis in Table 3.3. Without correction, the model-human correlations for the part concepts are: 0.403 for the set-based exemplar model, 0.505 for the transition exemplar model, 0.512 for the tree-based exemplar model, and 0.543 for the generative model (note that rank-order among the models does not change as a result of excluding these datapoints).

For the parameterized part example in Table 3.4, changing the argument uniformly, i.e. in all places where it occurs, leads to consistently higher scores than changing the argument differently in different places; however, this difference is not significant. This difference is expected if subjects inferred the true generative model, since changes to the argument require only one use of the noise process, whereas

nonuniform changes require many different nodes to be generated by the noise process. Future research needs to determine whether this effect is real, perhaps by manipulating the diversity of parameter arguments in the observations.

For the single recursion example in Table 3.4, changing the color of a few nodes within the recursion results in a significantly lower generalization. At the same time, a very similar manipulation does not result in a significant change in the generalization rating for the multiple recursion example. Intuitively, we sometimes see a change as destroying a very obvious pattern structure whereas at other times, the change in structure is not assumed to be relevant. Future research needs to characterize when subjects infer that such a pattern exists, and when they instead assume coincidence.

The comparison between the frequency based exemplar models and the two models that rely on tree structure in the observations makes clear that subjects do make use of the fact that the observations are structured in their generalization judgments. Furthermore, comparing the tree exemplar model to the true generative model that makes use of more abstract structure hints at the possibility that subjects are relying on recursive structure in the observations. The individual response patterns in the results of our exploratory experiment highlight ways in which both the exemplar-based model and the generative model can be improved to more closely reflect human generalization patterns.

3.4 Learning, revisited

In Section 3.2.2, we made the simplifying assumption that human learners recover the true generating concept. This was due to computational constraints: it is challenging to perform Bayesian inference over a hypothesis space that encompasses arbitrary

probabilistic programs. What if we *actually* want to learn generative concepts from data? In the following, we outline a computational approach to inducing probabilistic programs from data, and show anecdotal evidence that it can learn simple programs in the domain of little trees. For a more technical in-depth description of this approach, see Hwang et al. [39].

3.4.1 Bayesian model merging

We build on Bayesian model merging [81], a framework for searching the space of generative models in order to find a model that explains the observed data well. This search proceeds by applying a series of "merge" transformations to an initial model. These transformations are selected to maximize the posterior probability $P(M|D) \propto P(D|M)P(M)$ of the model M given data D. This process is initialized with a model that assigns high likelihood $P(D|M)$ to the data, but has a low prior $P(M)$ due to its complexity ("data incorporation"). For instance, when this process is used to learn grammars from data (as in [81]), an initial grammar is constructed by extending a working grammar with ad-hoc rules that generate the data, but that usually lead to bad generalization performance. The transformations define a space of simplified grammars, each of which is reachable by applying some sequence of transformations. Bayesian model merging explores this space using beam search, with the posterior probability $P(M|D)$ as the search objective. This procedure gradually simplifies this grammar so that it exhibits lower complexity and better generalization, at the cost of assigning somewhat lower probability to the observed data.

3.4.2 Bayesian program merging

When models are expressed as probabilistic programs, the search strategy (beam search) can remain the same as in Bayesian model merging; however, we need to adapt data incorporation strategy, search moves, and how we estimate the search objective, $P(M|D)$.

Our data are little trees, which can be represented as nested lists. Each tree consists of nodes, and each node has a size and color attribute, along with a list of child nodes. This specification forms an algebraic data type—that is, a compound data type built from other data types:

$$
\begin{aligned}
\langle\text{tree}\rangle &\models (\text{node}\ \langle\text{data}\rangle\ \langle\text{trees}\rangle) \\
\langle\text{trees}\rangle &\models \bot \mid\ \langle\text{tree}\rangle\ \langle\text{trees}\rangle \\
\langle\text{data}\rangle &\models (\text{data}\ \langle\text{color}\rangle\ \langle\text{size}\rangle) \\
\langle\text{color}\rangle &\models \textit{integer} \\
\langle\text{size}\rangle &\models \textit{integer}
\end{aligned}
$$

For example, one datapoint could be `(node (data 4 10) (node (data 5 5)))`.

Given data in this format, we can directly translate each datapoint into a program that returns this datapoint (by calling the corresponding sequence of constructors). Given a list `datapoints`, we can then construct the initial probabilistic program as `(uniform-choice datapoints)`, the program that returns one of the datapoints uniformly at random.

The length of this program is linear in the number of datapoints, hence this initial program tends to be rather complex. Moreover, it severely overfits: it never generates data that is not in the initial dataset. This is where program transformations come

in: step by step, we identify and factor out (approximately) repeated structure, which leads to lower complexity and more generalization, at the cost of a decrease in the probability that the program assigns to the initial dataset.

The first program transform we have explored is *abstraction*. Abstraction introduces new functions based on repeated syntactic patterns. For programs constructed using data incorporation, such syntactic patterns directly correspond to patterns in the observed data. Finding matching subexpressions and extracting them into a new function allows us to transform, for example, this program

```
(uniform-choice
  (node • (node • (node •) (node •)))
  (node • (node • (node •) (node •)))
  (node • (node • (node •) (node •))))
```

into the following more compact form:

```
(define (f x y)
  (node • (node • (node x) (node y))))
(uniform-choice (f • •) (f • •) (f • •))
```

This program, you may note, can be compressed further without loss. We can remove one of the arguments to `f`, since `x` and `y` take an on the same values in all of their instantiations:

```
(define (f x)
  (node • (node • (node x) (node x))))
(uniform-choice (f •) (f •) (f •))
```

We call the larger class of transforms that contains this particular rewrite *deargumentation*. Transforms in this class notice when instantiations of a function argument can be expressed compactly in terms of other values available in the same context,

and—when that is the case—remove the argument and replace its use with the inferred proxy definition. This idea is very general: besides removing variables with identical (or similar) instantiations, more sophisticated versions can replace values with random variables and induce stochastic recursion.

Finally, we need to compute $P(M)$ and $P(D|M)$ in order to assign a score to each of the programs in our search space. For $P(M)$, we use a program-length prior to bias the search towards smaller programs:

$$P(M) \propto e^{-\alpha \, \mathrm{size}(M)} \tag{3.3}$$

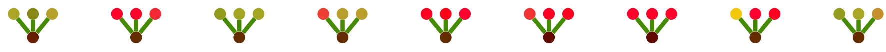
To estimate $P(D|M)$, we use a combination of Sequential Monte Carlo and Selective Model Averaging as described in Hwang et al. [39].

We have applied this scheme to a number of simple examples in the domain of little trees, as shown in Figures 3-3 to 3-6. For each example, we show (a) observations generated by a probabilistic program; data of this type constitutes the training set for our algorithm. We also show (b) samples from the induced program. To the extent that samples from the induced program look like samples from the original program, even when they are not identical, we have succeeded in capturing structure in the underlying generative process.

3.5 Conclusion

Most studies of concept learning have focused on relatively unstructured objects based on simple features. We have suggested viewing concepts as probabilistic programs that describe stochastic generative processes for more structured objects. In this view, concepts denote distributions over objects, and these distributions are built

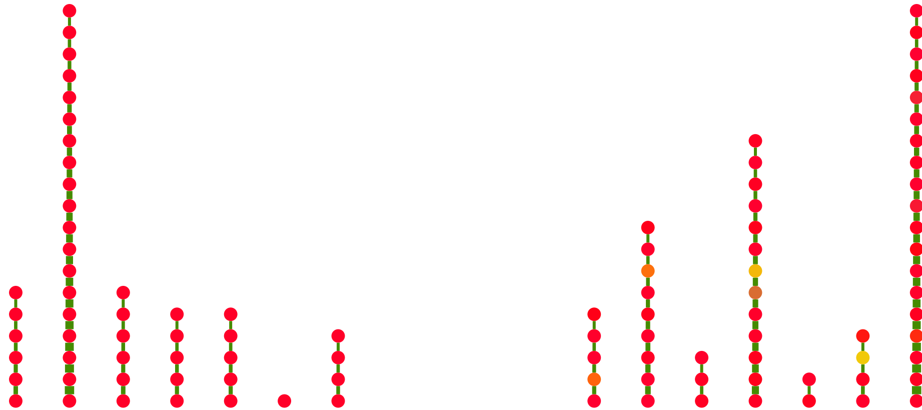
compositionally. We explored this idea within a domain of tree-like objects, carried out a study of human generalization using a broad variety of concepts in this domain, and sketched an algorithm for learning such concepts from data. Our results suggest that humans are able to extract abstract regularities, such as recursive structure, from examples, and that algorithms can extract similar regularities, but also that there are many subtle effects to be discovered and accounted for in such domains.



(a) Samples from the original program

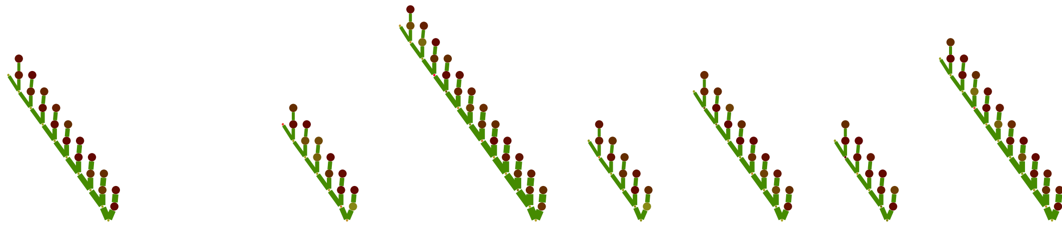
(b) Samples from the induced program

Figure 3-3: Program induction example: flower



(a) Samples from the original program (b) Samples from the induced program

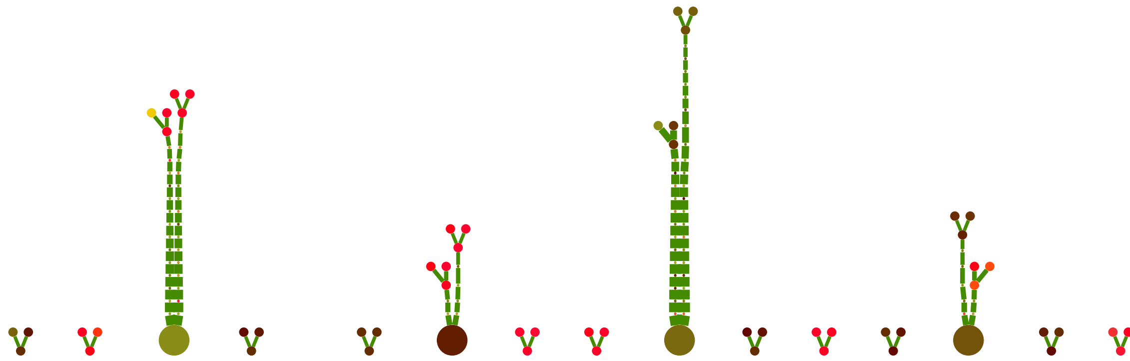
Figure 3-4: Program induction example: simple recursion



(a) A sample from the original program

(b) Samples from the induced program. A single observation is sufficient for the induction of a recursive program

Figure 3-5: Program induction example: vine



(a) Samples from the original program

(b) Samples from the induced program

Figure 3-6: Program induction example: tree

Chapter 4

Reasoning about Reasoning as Nested Conditioning

4.1 Introduction

Reasoning about the beliefs, desires, and intentions of other agents—*theory of mind*—is a central part of human cognition and a critical challenge for human-like artificial intelligence. Reasoning about an opponent is critical in competitive situations, while reasoning about a compatriot is critical for cooperation, communication, and maintaining social connections. A variety of approaches have been suggested to explain humans' theory of mind. These include informal approaches from philosophy and psychology, and formal approaches from logic, game theory, artificial intelligence, and, more recently, Bayesian cognitive science. Many of the older approaches neglect a critical aspect of human reasoning—uncertainty—while recent probabilistic approaches tend to treat theory of mind as a special mechanism that cannot

This chapter is based on Stuhlmüller and Goodman [83].

be described in a common representational framework with other aspects of mental representation. In this chapter, we discuss how probabilistic programming, a recent merger of programming languages and Bayesian statistics, makes it possible to concisely represent complex multi-agent reasoning scenarios. This formalism, by representing reasoning itself as a program, exposes an essential contiguity with more basic mental representations.

Probability theory provides tools for modeling reasoning under uncertainty: distributions formalize agents' beliefs, conditional updating formalizes updating of beliefs based on evidence or assertions. This approach can capture a wide range of reasoning patterns, including induction and non-monotonic inference. In cognitive science, probabilistic methods have been very successful at capturing aspects of human learning and reasoning [92]. However, the fact that conditioning is an operation *applied to* such models and not itself *represented in* such models makes it difficult to accommodate full theory of mind: We would like to view reasoning as probabilistic inference and reasoning about others' reasoning as inference about inference; however, if inference is not itself represented as a probabilistic model we cannot formulate inference about inference in probabilistic terms.

Probabilistic programming is a new, and highly expressive, approach to probabilistic modeling. A probabilistic program defines a stochastic generative process that can make use of arbitrary deterministic computation. In probabilistic programs, conditioning itself can be defined as an ordinary function within the modeling language. By expressing conditioning as a function in a probabilistic program, we represent knowledge about the reasoning processes of agents in the same terms as other knowledge. Because conditioning can be used in every way an ordinary function can, including composition with arbitrary other functions, we may easily express nested conditioning: we can condition any random variable, including random variables that

are defined in terms of other conditioned random variables. Nested conditioning describes reasoning about reasoning and this makes theory of mind amenable to the kind of statistical analysis that has been applied to the study of mental representation more generally.

The probabilistic program view goes beyond other probabilistic views by extending compositionality from a restricted model specification language to a Turing-complete language, which allows arbitrary composition of reasoning processes. For example, the multi-agent influence diagrams proposed by Koller and Milch [45] combine the expressive power of graphical models with the analytical tools of game theory, but their focus is not on representing knowledge that players' might have about other players' reasoning.

In the following, we show examples of how programs with nested conditioning concisely express reasoning about agents in game theory, artificial intelligence, and linguistics. In Chapter 5, we will describe the challenges in computing the predictions of these models, give a generic Dynamic Programming inference algorithm for probabilistic programs, and explain how it can help address some of these practical challenges.

4.2 Modeling theory of mind as nested conditioning

In Chapter 2, we outlined the probabilistic programming approach to capturing uncertain knowledge. We described how the operation of conditioning itself could be represented as an ordinary function in Church, `query` (Figure 2-3). This representation opens up an intriguing possibility: we can nest a call to the `query` function within other calls to this function. If we view `query` as capturing reasoning, then such nested models will capture reasoning about reasoning. We now illustrate how

```
(define (sample-location)
  (if (flip .55)
      'popular-bar
      'unpopular-bar))

(define (alice depth)
  (query
   (define alice-location (sample-location))
   alice-location
   (equal? alice-location (bob (- depth 1)))))

(define (bob depth)
  (query
   (define bob-location (sample-location))
   bob-location
   (or (= depth 0)
       (equal? bob-location (alice depth)))))
```

Figure 4-1: A Schelling coordination game in Church. Two agents, Alice and Bob, want to meet. They choose which bar to go to by recursively reasoning about one another.

this approach can be used to capture multi-agent reasoning, using examples from game theory, linguistics, and artificial intelligence.

4.3 Schelling coordination games

As a first illustration of our approach to theory of mind, consider a coordination game of the kind discussed by Schelling [74]: Two agents, Alice and Bob, want to meet, and they share the common knowledge that there are two possible meeting locations, one of them slightly more popular than the other. Each agent is modeled as making an inference about where it would be best to go. This can be formalized as a conditional distribution: "my location conditioned on my partner choosing the same location."

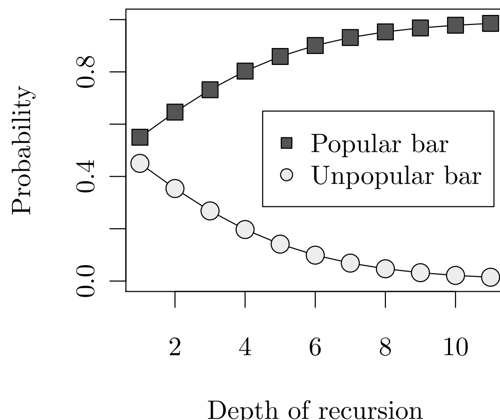


Figure 4-2: As the depth of recursive reasoning increases, the agents' actions converge to a focal point in the Schelling coordination game shown in Figure 4-1.

Since each agent starts with the bias that the other agent is more likely to go to the more popular place, they are yet more likely to go to this place themselves. This reasoning proceeds recursively, increasing the chance that the agents will go to the popular place. Each stage of this recursion is a conditional distribution.

This is an instance of *planning as inference*: we transform the problem of finding high-utility actions into the problem of computing a conditional distribution. The problem of maximizing expected utility can generally be translated into a likelihood maximization problem [94]. We are interested in modeling agents which choose only approximately optimally. This can be achieved using softmax-optimal decision-making, where an action a is chosen in proportion to its exponentiated expected utility under a belief distribution $P(s)$, i.e., $P(a) \propto \exp\left(\alpha \mathbb{E}_{P(s)}[U(a;s)]\right)$. We follow this approach throughout this chapter, however we simplify by using a single-sample approximation to estimate the expected utility, and assume a shared prior belief

distribution, except where we note otherwise.

Figure 4-1 shows how to use the planning-as-inference idea to formalize recursive reasoning in the Schelling coordination game as a Church program. The parameter `depth` controls the number of levels of recursive reasoning. Figure 4-2 shows how the marginal distribution changes as a function of this parameter. As the depth increases, the agents' actions converge on the focal point of the game—they always go to the popular location.

This illustrates a common pattern when modeling agents using probabilistic programs, namely the use of goal predicates instead of utility functions. It is possible to express choice in proportion to utility explicitly (by conditioning on a coin flip with a weight corresponding to the exponentiated utility of the outcome), but this is often equivalent to directly condition on the desired outcome (goal). This results in concise models that are intuitively appealing, since they match how we tend to think about our plans: not in terms of an explicit ordering on outcomes, but as aimed at achieving particular goals.

Uncertainty, including uncertainty about other players' uncertainty, can also be modeled using the standard tools of game theory, but the strengths of these tools lie elsewhere (e.g., amenability to analysis with respect to equilibria). In a discussion of games with incomplete information, Kreps [47] writes:

> "If we wanted something really complex, we could imagine that player 3 has an assessment concerning player 2's assessment of player 3's assessment of player 1's utility function. (If you think this is painful to read, imagine having to draw the corresponding extensive form.)"

Probabilistic programs make it easy to concisely express facts about other players' state of mind. For example, the coordination game can easily be modified to capture

```
(define (speaker access state depth)
  (query
   (define sentence (sentence-prior))
   sentence
   (equal? (belief state access)
           (listener access sentence depth))))

(define (listener speaker-access sentence depth)
  (query
   (define state (state-prior))
   state
   (if (= 0 depth)
       (sentence state)
       (equal? sentence
               (speaker speaker-access state (- depth 1))))))
```

Figure 4-3: Pragmatic reasoning in language understanding as recursively nested conditioning. For the full model specification, see Stuhlmüller and Goodman [83].

false beliefs [83]: Bob believes that Alice wants to meet him, and Alice knows this, but in fact Alice wants to avoid Bob. The compositional nature of probabilistic programs allows anything that can be expressed on its own to become the subject of others' reasoning, including agents' beliefs and reasoning steps. This is key for accurately modeling the interaction between agents who can represent other agents as intelligent reasoners.

4.4 Language understanding

Communication can be seen as a special case of decision-making in a multi-agent context: a speaker chooses utterances given an intended interpretation, and a listener chooses an interpretation given an utterance. We build on the *rational speech-act* theory of language understanding [16]: listeners model speakers as choosing their

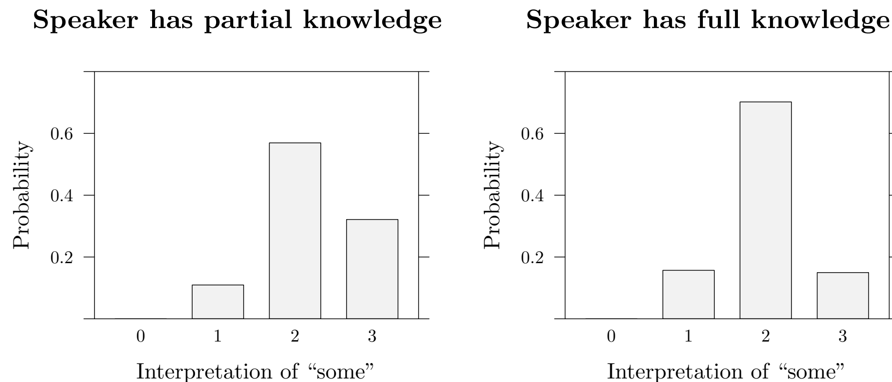


Figure 4-4: Predictions for the language understanding model shown in Figure 4-3. When the speaker has knowledge of the state of all 3 objects, the listener interprets "some" as likely to imply "not all". When the speaker has access to only 2 out of 3 objects, this implicature is cancelled and "some" is compatible with referring to all objects.

utterances approximately optimally based on social goals such as conveying information. Listeners then interpret utterances by "inverting" this model using Bayesian inference and drawing inferences about, among other things, the state of the world that caused the speaker to choose this utterance. This can again be seen as an instance of planning as inference, if we model the choice of utterances and interpretations as samples from conditional distributions.

The basic tenet of pragmatics is that listeners sometimes make inferences that differ from those that directly follow from the literal meaning of an utterance. Consider the sentence "some of the apples are red." In general, we take the speaker to mean that not *all* of the apples are red. This effect is called a scalar implicature [38]. The rational speech-act theory predicts that such effects are sensitive to the listener's beliefs about the speaker's knowledge of the state of the world. For example, if there are three apples in total, and if the speaker has only seen one of the apples (and it was red), then the speaker's utterance "some of the apples are red" does not

imply that not all of them are red. This effect has been confirmed experimentally in Goodman and Stuhlmüller [23].

Figure 4-3 shows one way to model speaker and listener in this situation. The speaker chooses a sentence conditioned on the listener inferring the intended state of the world when hearing this sentence; the listener chooses an interpretation conditioned on the speaker selecting the given utterance when intending this meaning. After a few iterations (determined by the `depth` parameter), this mutual recursion bottoms out and the speaker interprets the utterance literally, i.e., the speaker chooses an intended state conditioned on the given sentence being true of this state. Figure 4-4 shows model predictions that mirror the implicature-cancelling effect found in experiments—the probability that all three apples are red, after hearing "some of the apples are red," is much less when the speaker has seen all of the apples.

The predicted interaction between the listener's interpretation and the speaker's knowledge does not depend on the particular choice of words and intended meanings, but is a more general result of the fact that the speaker is modeled as choosing words based on expected utility. We can easily replace the set of words used in the model to derive predictions about pragmatic inferences in other contexts. We can also change the shared background knowledge to derive predictions for interpretations in other contexts, all without changing the core model of language understanding. This is an example of the particular kind of modularity that results from representing agents and their mental content as functional probabilistic programs: A priori, different model parts are independent and can thus easily be replaced with alternatives. A posteriori, i.e., as the result of conditioning, nontrivial dependencies between different model elements can arise, such as those between speaker's knowledge and listener's interpretation.

```
(define (exp-utility outcome player)
  (cond [(win? player outcome) 1.0]
        [(draw? outcome) 0.1]
        [else 0.01]))

(define (sample-action state player)
  (query
   (define action (action-prior state))
   (define outcome (sample-outcome state action player))
   action
   (flip (exp-utility outcome player))))

(define (sample-outcome state action player)
  (let ([next-state (transition state action player)])
    (if (terminal? next-state)
        next-state
        (let ([next-player (other-player player)])
          (sample-outcome next-state
                          (sample-action next-state
                             next-player)
                          next-player)))))

(define start-state
  '((0 o 0)
    (o x x)
    (0 o 0)))

(sample-action start-state 'x)
```

Figure 4-5: Tic-tac-toe in Church. Each player chooses actions by sampling an action, simulating the game until the end, and choosing actions in proportion to how likely they are to lead to a successful outcome at the end of the game. This mental simulation includes reasoning about both players' reasoning at all future game steps, including their reasoning about the other player. For the definition of functions such as `action-prior` and `transition`, see Stuhlmüller and Goodman [83]. To make multi-step planning more robust, adjust the optimization strength by sampling and conditioning on multiple outcomes in `sample-action`.

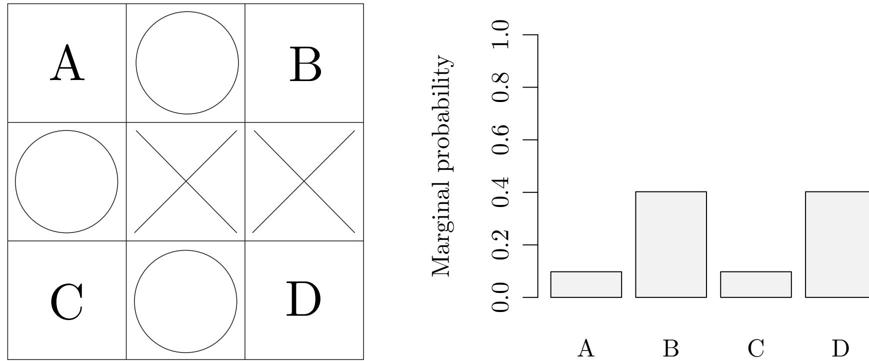


Figure 4-6: Where should **X** move next? This plot shows the predictions derived from the model in Figure 4-5. B and D are more likely to lead to a successful outcome, therefore they are more likely to be chosen.

4.5 Game playing

Theory of mind is an essential part of playing games such as chess and Go. Each player chooses moves depending on their beliefs about the other players, including their beliefs about the other players' beliefs. As in the Schelling coordination game, this kind of reasoning can be modeled using the planning-as-inference transformation: actions are chosen conditional on ultimately leading to a successful outcome. Whereas the Schelling coordination game is a one-shot setting, many games require sequential decision-making. Moves made in one turn depend on the expected consequences for subsequent turns, and on each player's decision-making strategy in subsequent turns.

As a simple example of a two-player game with sequential decision-making, consider Tic-tac-toe. Figure 4-5 shows the core of a Church implementation of a player. Figure 4-6 shows a simple prediction by this model that matches our intuitions: in a situation where **X** is not directly threatened and where **X** has two possible types of moves, one of which opens up a double threat, the forking move will be chosen, as it

allows **X** to win two turns from now.

The player implemented in Church recursively reasons about the future moves of another player who implements the same strategy. In the first round, player X samples an action conditioned on the game that starts with player X taking this action ultimately leading to a win for player X. Sampling from this conditioned distribution requires a sample from another conditioned distribution, namely from the distribution on player O's action in round two, which is conditioned on player O ultimately winning. This in turn depends on player X's action in round three, and so on, until the game tree bottoms out at a win for one of the players or at a draw.

The functions `sample-action` and `sample-outcome` make no mention of Tic-tac-toe in particular—they are a fully generic implementation of approximately optimal decision-making in a setting where two players take turns. By supplying different implementations of functions such `action-prior` and `transition`, we can use the same framework to model other games. This suggests that, to some extent, representation of players and games can be studied independently, with interesting predictions arising from their interaction. For example, from a psychological perspective, one can ask whether and when a player model that judges actions by sampling a single outcome matches empirical data, and when a more strongly optimizing model is more accurate.

Representing sequential decision-making in games as probabilistic programs naturally lends itself to interesting extensions. Instead of modeling both players as identical thinkers, we can model their particular strategic tendencies, their potentially approximate evaluation of game states, their beliefs about who they are playing, and their process of learning about the other player from past moves. This could be used, for instance, to build computer opponents that reason about the human player's state of mind and that use sophisticated tactics such as "misleading the player," not

because such tactics are built in, but as a consequence of rational planning with a model that represents the human player's model of the situation. This would have relatively little effect in a game like Tic-tac-toe, but an enormous effect in games like Go and Poker.

4.6 Induction puzzles

Induction puzzles are an instance of multi-agent reasoning that goes beyond the two-player case. Typically, induction puzzles describes a scenario involving multiple agents that are all assumed to go through similar reasoning steps. In such a scenario, the outcome can usually be determined inductively by first solving a simple case, then assuming that all agents know the solution to this simple case, which in turn makes another case simple to solve for all agents.

We consider a stochastic version of the *Blue-Eyed Islanders* puzzle (also known as the *muddy children* and *cheating husbands* problem), a well-known problem in epistemic logic [19, 89]. The setup is as follows: There is a tribe on a remote island. Out of the n people in this tribe, m have blue eyes. Their religion forbids them to know their own eye color, or even to discuss the topic. Therefore, everyone sees the eye color of every other islander, but does not know their own eye color. If an islander discovers their eye color, they have to publicly announce this the next day at noon. All islanders are highly logical. One day, a foreigner comes to the island and—speaking to the entire tribe—he truthfully says: "At least one of you has blue eyes." What happens next?

Intuitively, the solution is as follows. If there is only one islander with blue eyes, the islander will see that no other person has blue eyes and will announce their knowledge the next day. If no islander does so the next day, then everyone knows

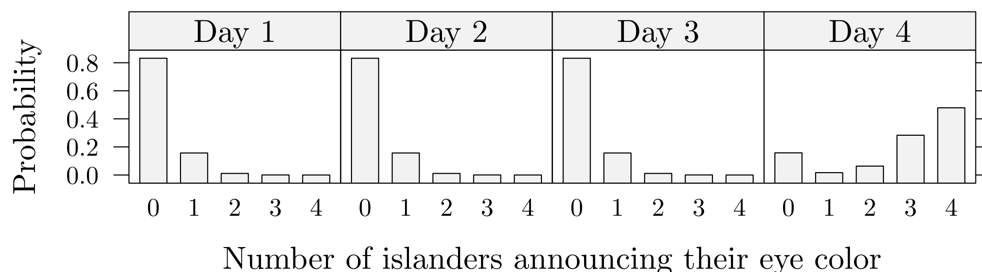
```
(define (agent t raised-hands others-blue-eyes)
  (query
   (define my-blue-eyes (if (flip baserate) 1 0))
   (define total-blue-eyes (+ my-blue-eyes others-blue-eyes))
   my-blue-eyes
   (and (> total-blue-eyes 0)
        (! (λ () (= raised-hands
                    (run-game 0 t 0 total-blue-eyes)))
           2))))

(define (get-raised-hands t raised-hands true-blue-eyes)
  (+ (sum-repeat
       (λ () (agent t raised-hands (- true-blue-eyes 1)))
       true-blue-eyes)
     (sum-repeat
       (λ () (agent t raised-hands true-blue-eyes))
       (- num-agents true-blue-eyes))))

(define (run-game start end raised-hands true-blue-eyes)
  (if (>= start end)
      raised-hands
      (run-game
        (+ start 1)
        end
        (get-raised-hands start raised-hands true-blue-eyes)
        true-blue-eyes)))
```

Figure 4-7: Church implementation of a stochastic version of the Blue-Eyed Islanders puzzle. For the full specification, see Stuhlmüller and Goodman [83].

‘At least one of you has blue eyes.’



‘At least one of you has blue eyes and a twitchy hand.’

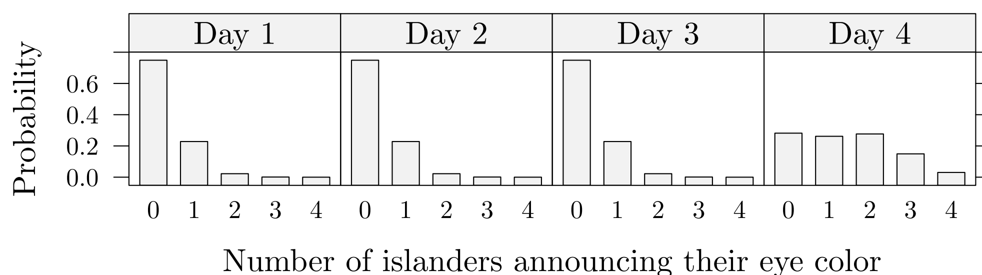


Figure 4-8: Model predictions for a stochastic version of the Blue-Eyed Islanders puzzle with population size 4, all islanders blue-eyed. Four days after the foreigner makes his announcement, the islanders are likely to realize that they have blue eyes. However, if the foreigner (truthfully) states that one of the blue-eyed islanders has a twitchy hand and mistakenly announces that she has blue eyes 10% of the time, this inference becomes much less pronounced.

that there are at least two islanders with blue eyes. Since each of the two islanders with blue eyes only observes one other blue-eyed islander, they can deduce that they must have blue eyes themselves, and so the two blue-eyed islanders announce their eye colors on the second day. Generalizing this line of reasoning, all m blue-eyed islanders announce their eye color on the m-th day.

Figure 4-7 shows a formalization of a stochastic version of this puzzle. We have converted a multi-agent sequential planning problem into an inference problem that uses nested conditioning to model recursive reasoning. In this model, we have increased the optimization strength of action selection in softmax-optimal sampling (α) by conditioning on two successes instead of one, thus moving part of the way from probability matching to utility maximization.

Figure 4-8 shows the corresponding model predictions for population size 4 when all islanders are blue-eyed: on the 4th day, it is most likely that all islanders decide to announce their eye color. In this version, the probability of an islander making the announcement is proportional to their estimate for how likely they are to have blue eyes.

The induction that happens as a result of this conversion differs from the backwards-induction in the previous section. In playing games, the current action is chosen based on reasoning about the likely future outcome of the game, which depends on modeling the other agents' *future* reasoning. In this induction puzzle, the current action is chosen based on current beliefs ("do I have blue eyes or not?"), which are the result of explaining past observations ("how many islanders have made the announcement?") by reasoning about other agents' *past* reasoning.

One of the main attractions of probabilistic programming as a modeling framework is that it is easy to rapidly prototype complex probabilistic models, since implementations of probabilistic programming languages provide *generic* inference

algorithms; it is not necessary to design model-specific algorithms. This is highly useful in multi-agent scenarios, where complex interdependencies between agents' conditional inferences can make it difficult to "see" what the outcome of a model change is. For example, what if the foreigner had said to the islanders: "At least one of you has blue eyes, and raises their hand by accident 10% of the time." Changing the model to account for this requires one additional line of code (see Stuhlmüller and Goodman [83]). The predictions are shown in Figure 4-8. It still takes four days for the islanders to realize with some uncertainty what their eye color is, but each islander is less certain and therefore less likely to make the announcement.

The idea of common knowledge is central to both versions of this induction puzzle. Before the foreigner makes his announcement, every islander already knows that there is at least one other islander with blue eyes simply by observing the others' eye colors. By speaking in front of the entire tribe, the foreigner causes this fact to become common knowledge: now, everyone knows that everyone knows this fact, and everyone knows that everyone knows that everyone knows, and so on. This idea is reflected in the Church program by the condition `(> total-blue-eyes 0)`. This condition applies both to the islander whose reasoning is modeled, to his mental model of other islanders' reasoning, to the mental models of their mental models, and so on. If we remove this condition, the effect of all islanders inferring their eye color on the m-th day goes away. By representing this multi-agent reasoning scenario as a probabilistic program, we have formalized a hypothesis about what it means to have common knowledge, and we are able to explore the implications of this hypothesis.

4.7 Discussion

Our approach to theory of mind is based on a long history of both informal and computational accounts of theory of mind [e.g., 7, 25, 29, 96]. For an outline of existing logical and probabilistic approaches from the perspective of building cognitive architectures, see Bello [6]. While much of the relevant work on cognitive architectures aims to take into account resource constraints of the agents that are modeled, this is not our goal; we have presented tools for *computational-level modeling* of agents' reasoning in multi-agent scenarios. Such models constitute a normative standard that can be compared to actual patterns in human reasoning, which in turn may help us devise process-level descriptions of the same phenomena.

Most directly, we build on the Bayesian approach to theory of mind [4, 5], which takes the *principle of rational action* to be central to the concept of intentional agency: all else being equal, agents are expected to choose actions that satisfy their goals as effectively as possible. Given a probabilistic generative model of an agent's planning process that has this property, and given observations, Bayesian theory of mind simply proposes that we can infer the hidden internal states of the agent—beliefs, desires, and goals—by inverting this model, i.e., by conditioning. By viewing conditioning as a function that is represented in a probabilistic program, we have presented a simple, unified formalism for implementing Bayesian theory of mind, and for extending it to settings such as the recursive reasoning scenarios presented in this chapter.

Philosophically, our approach suggests that it may be possible to acquire the mental machinery for theory of mind from simpler primitives such as control structure, random choice, and recursive functions. Specialized mental modules—such as a "belief box" and a "desire box" [61]—might arise simply as a consequence of an

agent's attempt to explain her observations in a parsimonious way, or might not turn out to be necessary to explain our cognitive capabilities in the first place.

4.8 Conclusion

We have described how probabilistic programs can represent multi-agent scenarios by modeling each agent's reasoning as conditional sampling. Since conditioning is an operation that can be defined *within* such models, it can become the subject of other agents' reasoning. This representation goes beyond existing formalisms, such as graphical models, where conditioning is more naturally seen as an operation that is *applied to* a model, but not itself represented.

There are many research opportunities related to this way of studying theory of mind. One avenue is to translate existing models used in fields such as game theory and linguistics into the framework of probabilistic programs. Extensions of these models that are difficult to express using existing modeling languages, but that are natural using this new approach, will suggest themselves. Further, integration of diverse aspects of social cognition will be fostered by representing them in a common representational system—libraries of probabilistic program components for cognition can be naturally integrated into a more complete architecture.

Viewing theory of mind as reasoning about reasoning and formalizing this as nested conditioning is appealing on both philosophical and scientific grounds. By realizing theory of mind in a uniform representational framework for reasoning under uncertainty, we expose essential assumptions and provide opportunities for constructing more complex, and realistic, models of social reasoning.

Part II

Algorithms

Chapter 5

Dynamic Programming for Probabilistic Programs

5.1 Introduction

Probabilistic programming allows rapid prototyping of complexly structured probabilistic models without requiring the design of model-specific inference algorithms. This makes probabilistic programs attractive for scientific research: when hypotheses are formalized as programs, it is possible to quickly explore the space of hypotheses. The same features make probabilistic programs compelling for education: students can focus on understanding modeling and inference *patterns* before they need to learn about inference *implementations*.

However, the performance of current inference algorithms for generic probabilistic programs can vary greatly between models, even for models with a very small number of random choices. This presents an obstacle to the use of probabilistic programs

This chapter is based on Stuhlmüller and Goodman [82] and Stuhlmüller and Goodman [83].

in research and teaching. In fact, many of the models used in these domains are small enough that exact computation is feasible in principle, but they often exhibit patterns, such as nested conditioning, that make naive enumeration intractable.

In this chapter we develop a generic Dynamic Programming algorithm, which expands the applicability of exact inference for probabilistic programs. Given an interpreter for an arbitrary probabilistic programming language and a discrete probabilistic program, this algorithm computes the marginal distribution of the program—i.e., its distribution on return values—while sharing subcomputations where possible. By viewing conditioning as marginalization of a rejection sampler, this captures the full range of probabilistic operations over arbitrary models.

The key obstacle to Dynamic Programming, which is neither present in caching deterministic interpreters nor in Dynamic Programming algorithms for more restricted model classes, is the possibility of stochastic self-recursion: an interpreter call with particular arguments can result in a call with the same arguments. Figure 5-1 shows a program that exhibits this property. This is not a corner case: for instance, all models that implement conditioning via rejection sampling have this property (Figure 2-3).

To make Dynamic Programming possible in the presence of recursion, we first compile the given probabilistic program to an intermediate representation that reifies dependencies between sub-distributions. We then compute the marginal distribution from this representation. Our intermediate representation is a generalization of sum-product networks [67] that makes dependencies—including recursive dependencies—explicit: a *factored sum-product network* (FSPN). While computing the distribution implied by a sum-product network is linear in the size of the network, FSPNs are more difficult to solve in general. We solve FSPNs by clustering their vertices into strongly connected components and by solving each component using fixed-point

```
(define (game player)
  (if (flip .6)
      (not (game (not player)))
      (if player (flip .2) (flip .7))))

(game true)
```

Figure 5-1: A simple recursive probabilistic program

iteration.

In the following, we first describe the structures our algorithm operates on: probabilistic programs, their interpreters, and FSPNs. We then present the two steps of our algorithm, compilation of programs to FSPNs and computation of marginal distributions given a FSPN. We demonstrate the algorithm on examples used in teaching, cognitive science research, and game theory, and explain what makes it attractive in each case. We relate the algorithm to the literature and conclude with future research directions.

5.2 Inference as marginalization

Recall that a probabilistic program is a program in a language with primitives for sampling from distributions such as Bernoulli and multinomial. Probabilistic programs describe generative models and thus denote distributions. An interpreter specifies this denotation by implementing a process that, given a program, generates samples from the program's distribution. For example, an interpreter for the Church language [26] takes a program expression and environment, and returns a sample from the program's distribution on Church values. This sample is generated using recursive calls to the interpreter, with each subcall defining a distribution on values and resulting in a sample from this sub-distribution.

The problem of inference for generative models is commonly formulated in terms of a conditioning. However, as we have seen in Chapter 2, for any conditional distribution there is an equivalent unconditioned model that samples outcomes with the same probabilities. Inference can be understood as the problem of marginalization of a model that contains a call to the `rejection-query` function (Figure 2-3). Of course, actually drawing conditional samples using `rejection` is very inefficient: In general, we may have to tolerate an exponential number of rejected samples before the condition is satisfied. However, if we could efficiently marginalize the `rejection` procedure, eliminating all zero-probability paths, then we would have solved our target inference problem.

For many probabilistic programs, efficient marginalization of this sort is possible. We are interested in programs for which many different executions share substructure. Problems with this character are classically amenable to Dynamic Programming. Recursive programs, like `rejection-query`, which involve multiple executions of the same procedure application, provide a particularly rich opportunity to exploit shared substructure. We will focus on these cases in the examples below.

5.3 Multiply-intractable distributions

As seen in Chapter 4, probabilistic programs that use stochastic recursion to express *nested* conditioning can represent a wide and flexible space of multi-agent reasoning. We now explain the challenges of computing (or sampling from) the distributions defined by models with nested conditioning, and then sketch a Dynamic Programming algorithm that addresses some of these challenges.

Instead of directly attempting to sample from a program's distribution on return values, consider the goal of sampling from a program's distribution on executions.

```
(query
  (define a (sample-integer 10))
  (define b
    (query
      (define c (sample-integer 10))
      c
      (> (+ a c) 8)))
  a
  (= (+ a b) 13))
```

Figure 5-2: A simple program with nested conditioning.

An execution corresponds to a complete sequences of random choices and determines a return value. For an unconditioned program, the probability of an execution is the product of all random choices that occur within this execution, and thus is easy to compute. For a conditioned program, the probability of an execution is only proportional to this product, since the condition rules out some executions, redistributing their mass on the "allowed" executions. Computing this probability exactly is often intractable, as it requires integrating over all program executions. Many algorithms for approximate inference require only the unnormalized probability (i.e. the probability up to an unknown normalizing constant).

However, in the setting of nested conditioning, even the unnormalized probability of an execution can be difficult to compute.

For example, consider the program in Figure 5-2. This program could model the following situation: Two agents play a game in which each agent needs to name a number between 0 and 9 and they win if their numbers add up to 13. The first player knows this, and he knows that the second player gets to see the number the first player chooses, but the second player mistakenly thinks that the two win if their numbers add up to any number greater than 8 (and the first player knows this as

well). What number should the first player choose?

In this game, we can write the probability of a program state as follows:

$$
\begin{aligned}
p(a, b | a + b = 13) &= \frac{p(a)p(b|a)\delta_{a+b=13}}{\sum_{a',b'} p(a')p(b'|a')\delta_{a'+b'=13}} \\
&\propto p(a)p(b|a)\delta_{a+b=13}
\end{aligned}
$$

Here, $\delta_{a+b=13}$ is the delta function that returns 1 if $a + b = 13$ is satisfied, otherwise 0. The distribution $p(b|a)$ is itself defined in terms of a conditional distribution q:

$$
p(b|a) = q(b|a, a + b > 8) = \frac{q(b|a)\delta_{a+b>8}}{\sum_{b'} q(b'|a)\delta_{a+b'>8}}
$$

Making use of the fact that $p(a)$ and $q(b|a)$ are uniformly distributed, the *unnormalized* probability of a program state is

$$
p(a, b | a + b = 13) \propto \frac{\delta_{a+b=13}}{\sum_{b'} \delta_{a+b'>8}}.
$$

Intuitively, this means that the probability of a state (a, b) is inversely proportional to the number of assignments b' that the second player could have chosen to make the sum $a + b'$ greater than 8, since each such assignment reduces the chance that the second player chooses the assignment that makes $a + b = 13$ true. The first player is best off choosing $a = 4$, such that for the second player, the goal of making their sum greater than 8 coincides as much as possible with the goal of making their sum equal to 13.

For the purpose of inference, the relevant insight is that computing the unnormalized probability requires us to sum over the state space of the inner query. That is, even the unnormalized probability of the outer query depends on the normalizing

constant of the inner query. While this is easy to compute for the given toy problem, it is typically difficult and makes inference in models with nested queries challenging.

To generalize this line of reasoning, assume that we are interested in sampling from a distribution

$$p(y|c_1) = \frac{p(y)\delta_{c_1(y)}}{\int p(y)\delta_{c_1(y)}\,\mathrm{d}y} \propto p(y)\delta_{c_1(y)}.$$

Here, y is a program state, c_1 is a condition on that state, $p(y)\delta_{c_1(y)}$ is the unnormalized probability of y, and $Z_y = \int p(y)\delta_{c_1(y)}\,\mathrm{d}y$ is the normalization constant.

Suppose that the distribution on program states $p(y)$ factors as follows:

$$p(y) = p(y_1, y_2) = p(y_1)p(y_2|y_1)$$

Assume further that $p(y_2|y_1)$ is defined in terms of a conditional distribution itself, i.e., we are describing a distribution defined in terms of a query within a query[1]:

$$\begin{aligned} p(y_2|y_1) &= q(y_2|y_1, c_2) \\ &= \frac{q(y_2|y_1)\delta_{c_2(y_2)}}{\int q(y_2|y_1)\delta_{c_2(y_2)}\,\mathrm{d}y_2} \end{aligned}$$

Then, the *unnormalized* probability of a state y is:

$$\begin{aligned} p(y|c_1) &\propto p(y_1)p(y_2|y_1)\delta_{c_1(y)} \\ &= \frac{p(y_1)q(y_2|y_1)\delta_{c_2(y_2)}\delta_{c_1(y)}}{\int q(y_2|y_1)\delta_{c_2(y_2)}\,\mathrm{d}y_2} \end{aligned}$$

[1]Note that in the term $q(y_2|y_1, c_2)$ there is an ambiguity: y_1 is a parameter to this function, while c_2 is a condition. This is a common ambiguity in probability notation that is clarified in probabilistic program notation.

Note that the denominator depends on y_2 and thus on y. It affects the relative probabilities of different y and cannot be ignored, even if we are only interested in the unnormalized probability. As a consequence, evaluation of unnormalized $p(y|c_1)$ requires integrating over the domain of y_2, which is often intractable. Since inference methods such as MCMC are used to approximately sample from distributions that are "intractable" in the sense that we cannot compute the normalization constant, models with nested conditioning are called *doubly-intractable*.

Learning the parameters of a graphical model is an example of such a problem: for each parameter setting, we need to integrate over the space of all explanations of the data in order to compute a value proportional to the likelihood of the data. While auxiliary variable techniques can make sampling tractable in special cases [60], Murray and Ghahramani [59] conjecture that for general undirected models, there exists no tractable MCMC scheme that results in the correct stationary distribution.

Most of the examples we have seen define a nesting of conditional distributions of depth greater than 2. For such *multiply-intractable* distributions, the difficulty of evaluating f increases exponentially in the depth. If we assume that $q(y_2|y_1)$ factors into an unconditioned and a conditioned distribution, and if we reason analogously to the steps above, we conclude that naive computation of the unnormalized probability of a single program execution y requires us to solve a multiple integral, thwarting existing methods for approximate inference.

5.4 A Dynamic Programming algorithm

Since existing methods of approximate inference are intractable for models with nested queries, we instead pursue exact inference methods that exploit shared subcomputation to perform tractable inference for modestly sized state spaces. All of

the model predictions shown in Chapter 4 have been computed using this algorithm. We present this Dynamic Programming algorithm as a practical tool for modeling, not as an hypothesis about human processing; the structural insights this algorithm leverages, however, may prove useful in future exploration of the processes of social cognition. We next sketch this inference technique, then follow up with more technical details.

5.4.1 Approach

Our starting point is this: we are given an interpreter for a probabilistic programming language and a probabilistic program, and we want to compute the marginal distribution of this program. In other words, we want to compute the distribution on return values that we would sample from if we executed the program using this interpreter.

We assume that the interpreter is functional and defined recursively, i.e., in the process of evaluating a particular program, it calls itself and thus reduces the overall problem of evaluation to the problem of evaluating a number of smaller objects. Each of these subproblems has a unique distribution on solutions, and in the process of marginalization, some subproblems may occur multiple times. Our algorithm is based on the idea that we can detect repeated, identical subproblems and that we can then solve each subproblem only once, reusing the results for future occurrences of the same problem.

In Church, subproblems correspond to subexpressions with environments. For example, computing the marginal distribution of the Church program `(and (flip) (or (flip) (flip)))` involves computing the distribution of `(flip)` and of `(or (flip) (flip))`. In the setting of nested conditioning, the problem of computing

the marginal distribution of a query that contains other queries may be such that the same inner query can be reused for many different settings of the parameters of the outer query.

To understand what makes marginalization with reuse of computation challenging, compare to simplified versions of the problem. If we were trying to ensure reuse of repeated subcomputations for a deterministic interpreter, we could simply memoize it. For a stochastic language, we could imagine a similar approach: use the interpreter to recursively compute and cache the distribution for each unique interpreter call, computing all subdistributions required by a call before we compute the distribution of the call itself. However, consider the program shown in Figure 5-3. In order to compute the distribution of the first `if`-branch of `(game true)`, we need to know the distribution of `(game (not true))`, but this in turn depends on the distribution of `(game true)`. These *self-recursive* dependencies make direct caching impossible, as it would lead to infinite regress. A similar pattern is present in queries that are defined via rejection sampling.

Our algorithm addresses this challenge by first transforming the program into an intermediate structure that makes these dependencies explicit, then computing the marginal distribution from this structure in a way that is sensitive to the dependencies.

We first describe the interface the interpreter needs to satisfy such that we can use it to acquire sufficient information about the program to build our intermediate structure. We then present this structure and the steps of the algorithm.

By default, an interpreter takes a program and evaluates it, finally returning a single value. In order to control the evaluation process in a way that lets us acquire information about the structure of a program's distribution, we require that the interpreter is a *coroutine* that interrupts its evaluation and returns its current state

whenever it (1) makes a recursive subcall, (2) samples a primitive random choice, and (3) returns a terminal value. Each of these cases will correspond to a particular way of building structure in our intermediate representation.

For our intermediate representation, we desire that it factors out all "deterministic" computation, leaving only probability calculations, i.e., sums and products, and that it makes explicit the dependencies between the distributions resulting from subcomputations. Sum-product networks [67] satisfy the former requirement. We extend the formalism to *factored* sum-product networks in order to satisfy the latter: A *factored sum-product network* (FSPN) over variables $x_1, \ldots, x_d$ is defined as a directed graph with a uniquely labeled root node r. The internal nodes are sums and products. The leaves are indicators $x_1, \ldots, x_d$ and $\bar{x}_1, \ldots, \bar{x}_d$, and reference nodes $(y, \vec{x})$, where y is another node and $\vec{x}$ a vector of indicator values. Each edge (i, j) from a sum node i has a non-negative weight w_{ij}.

Let $\mathrm{Ch}(y)$ denote the children of node y. The value $\mathcal{V}(y, \vec{x})$ of a node y is defined as $\sum_{z \in \mathrm{Ch}(y)} w_{yz} \mathcal{V}(z, \vec{x})$ if y is a sum, as $\prod_{z \in \mathrm{Ch}(y)} \mathcal{V}(z, \vec{x})$ if y is a product, as $\mathbb{1}_{\vec{x}_j = x_j}$ if y is an indicator x_j, and as $\mathcal{V}(z, \vec{w})$ if y is a reference $(z, \vec{w})$. We denote the factored sum-product network F as a function of the indicator variables $\vec{x} = (x_1, \ldots, x_d, \bar{x}_1, \ldots, \bar{x}_d)$ by $F(\vec{x}) = \mathcal{V}(r, \vec{x})$. For any given $\vec{x}$, the value of the FSPN is the solution to the system of equations $F(\vec{x})$ (if a unique solution exists).

We use FSPNs to describe marginal distributions and indicators to query the probabilities of marginal values. To simplify notation, we write indicators as $\mathbb{1}_v$, denoting the x_i corresponding to value v. We write reference nodes as $r\!:\!v$, denoting $(r, [0, 0, \ldots, 0, 1, 0, \ldots, 0])$, where the only indicator entry that is 1 corresponds to value v. Negated indicators $\bar{x}_i$ are not used by our algorithm.

In the following, we refer to "root nodes" in addition to the node types introduced above. These nodes always have exactly one child and can hence be of either type sum

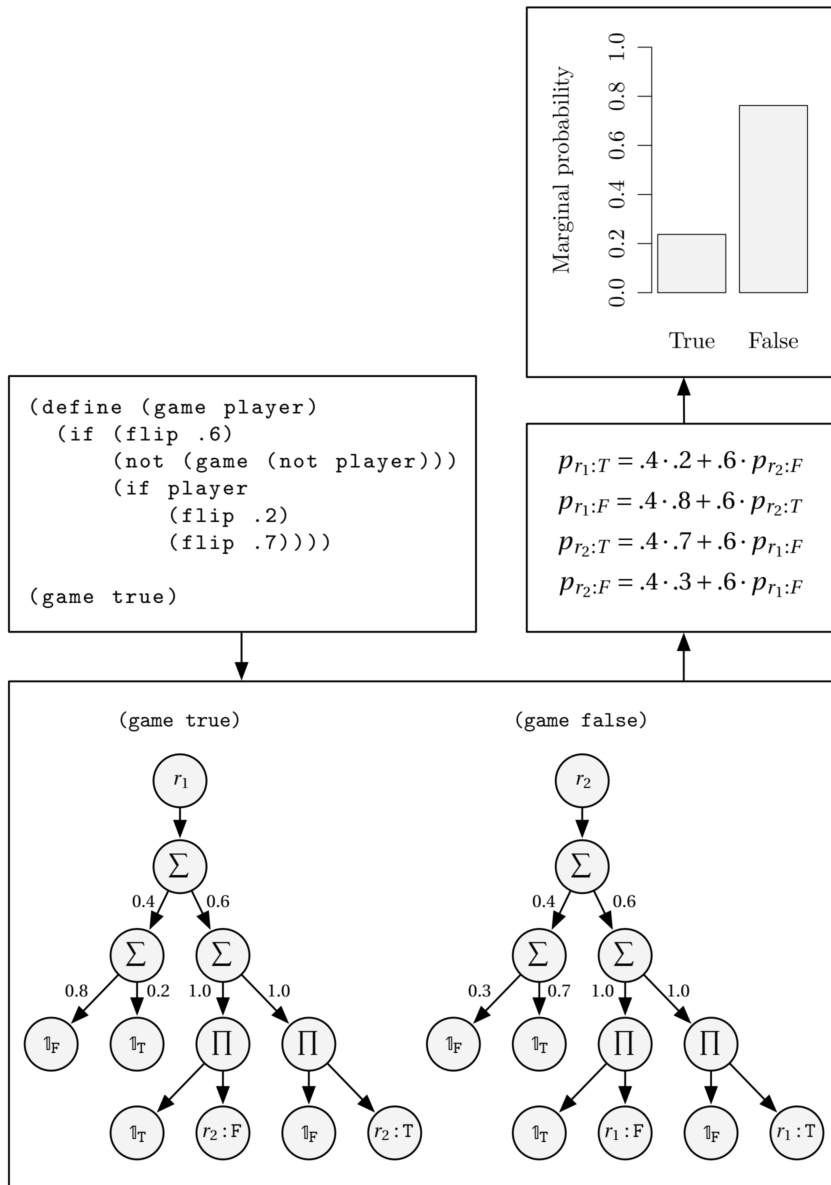


Figure 5-3: An example of applying the Dynamic Programming algorithm to a simple Church program. The program is compiled to a factored sum-product network, which directly corresponds to a system of equations that can be solved to infer the marginal distribution.

or product. The networks we build are sets of trees. Each node will be associated with a unique (ancestor) root node. Root nodes will correspond to subproblems. One of these root nodes is the root node referred to in the definition of a FSPN; it corresponds to the overall problem of computing the marginal distribution of the starting state of the given probabilistic program. In the following illustration of the process our algorithm uses to build a FSPN, we depict in addition to the previously built node n_{prev} and the weight of the edge from this node to the current node, w_{prev}, the root node r that is associated with the previous and current node.

5.4.2 Algorithm

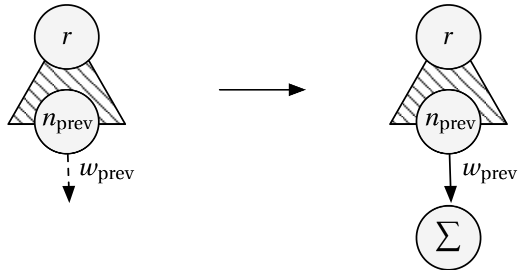
The algorithm proceeds in three steps. Figure 5-3 shows the result of applying each of the steps to a simple self-recursive program.

1. Compile program to FSPN.

We initialize the FSPN with a single node r. Our algorithm maintains a queue of tasks, each of which is a tuple of a thunk f (a function without arguments), a previous node n_{prev}, and an edge weight w_{prev} (a probability). The queue is initialized to a task for the first interpreter call, $(\lambda.\mathcal{I}(s), r, 1)$. While the queue is not empty, the algorithm takes the first task in the queue and evaluates the function call $f()$. There are three types of values that this function call can return: random choices, terminal values, and subcalls. We manipulate the FSPN and queue depending on the type of this value:

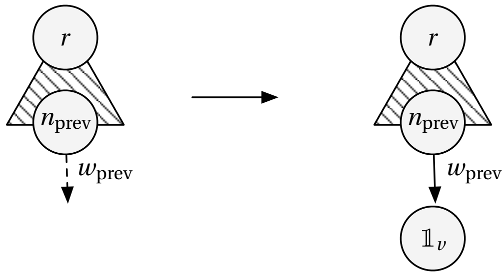
A **random choice** is a tuple $(c, \vec{v}, \vec{p})$ of a continuation c, a list of values $\vec{v}$, and a list of probabilities $\vec{p}$. The continuation c is a formal object that captures the current state of the computation; it can be called with a value to resume evaluation. $\vec{v}$ and $\vec{p}$ specify the distribution of the random choice where we interrupted evaluation. We

modify the FSPN by connecting a sum node $\sum$ to n_{prev} using w_{prev}:



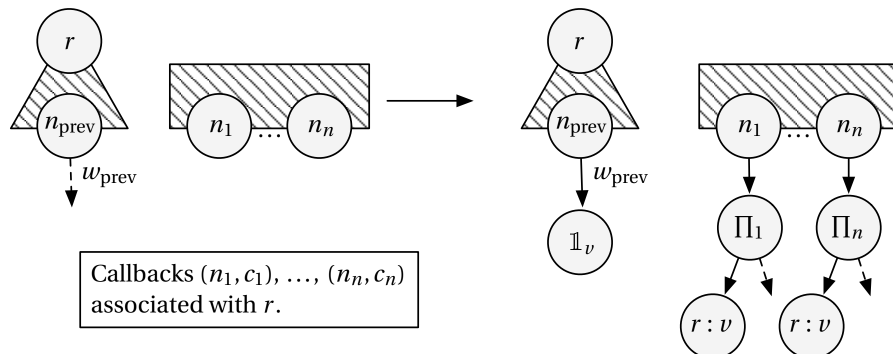
For each value-probability pair, we then add to the queue the task of exploring $c(v)$ with previous node $\sum$ and edge weight p. This reflects the fact that the marginal probability of any value under the current subproblem r is the sum of the probabilities of this value being returned for each of the ways of proceeding from the current program state, weighted by the probability of choosing each such way.

A **terminal value** v is treated depending on whether it is a new value for current root node r or whether it has been encountered before. If it has been encountered before, we simply add an indicator node for this variable to the FSPN, reflecting that for a program state that directly returns value v, the marginal probability is 1 for this value and 0 otherwise:



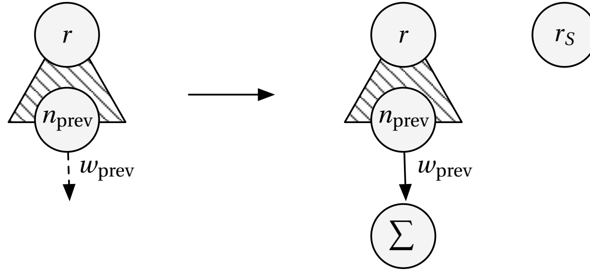
If we encounter v for the first time under root node r, this means that we just learned something about the set of support values of the subproblem corresponding to r. If this subproblem is used elsewhere, new parts of the program's state space just

became accessible to us. To store where a subproblem is used, we associate a list of *callbacks* with each root node. A callback is a pair of a node n (the node corresponding to the program state that referred to r's subproblem) and a continuation c (for proceeding from that program state, given a return value for r's subproblem). Given a new value v, we can build graph structure and store a queue entry for each such callback, reflecting that it is possible to continue from n with the marginal probability of v under r. For every callback (n_i, c_i), we add a product node $\prod_i$ and a reference node $r : v$ under node n_i, and add to the queue the task of exploring $c_i(v)$ with previous node $\prod_i$:



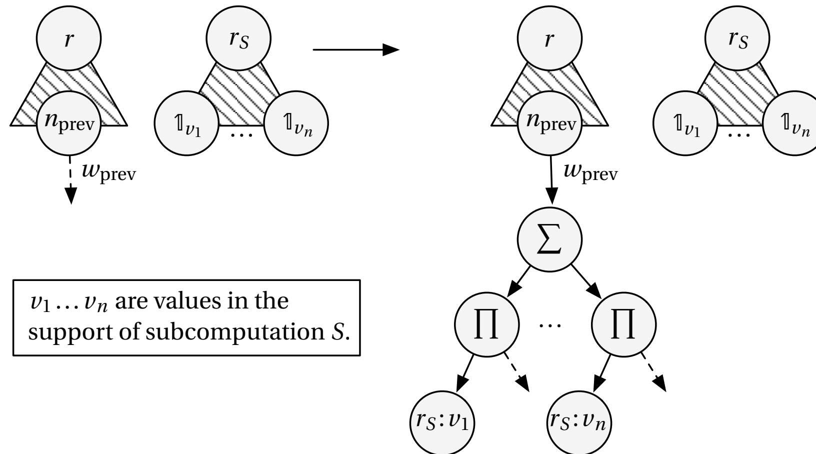
A **subcall** is a pair of a continuation c and an interpreter argument s. This is where we share repeated subcomputations: depending on whether we have seen s before or not, we proceed to explore this subcall and build graph structure, otherwise we simply refer to existing structure.

If we encounter s for the first time, we connect a sum node $\sum$ to n_{prev}, reflecting that the probability of any marginal value under r will be a sum of the probability of this value under all ways of continuing using different return values from subproblem s, weighted by the probability of these return values. We also add a new root node r_S to the graph which represents the subproblem s and its marginal distribution:



We add to the queue the task of exploring the interpreter call $\mathcal{I}(s)$ with previous node r_S, and add $(\sum, c)$ to the callbacks of r_S such that graph structure under $\sum$ will be built if marginal values for $\mathcal{I}(s)$ are found.

If we have seen the interpreter argument s before, we also add a sum node $\sum$ under n_{prev}, but then look up the existing root node r_S for subproblem S. If marginal values for this subproblem are already known, we can immediately build graph structure under $\sum$ that refers to the marginal probability of these values, and we can explore the current continuation c using these values. For each known return value v_i supported by S, we build a product node $\prod_i$ and reference node $r_S : v_i$ under $\sum$ and add to the queue the task of exploring $c(v_i)$ with previous node $\prod$.



As before, we also add $(\sum, c)$ to the callbacks of r_S in order to continue building graph structure under $\sum$ if more marginal values of r_S are found in the future.

This process terminates once the queue is empty. Alternatively, it is possible to bound the graph size, which results in lower bounds on the true marginal probabilities.

2. Convert FSPN to equations.

Following the value equations given in Section 5.4.1, we can directly convert a factored sum-product network into a system of polynomial equations. In addition, it is sometimes possible to reduce the time spent in the next step by applying a simple substitution-based equation simplifier.

3. Solve equations to get marginal distribution.

For probabilistic programs, the generated system of equations tends to be sparse, reflecting the fact that most interpreter calls that occur within the possible executions of a program do not depend on most other calls. We therefore cluster the equations into strongly connected components and solve the clusters of equations in topological order. Computing a topological order of strongly connected components is linear in the size of the graph [91]. By solving in topological order we know that all probabilities required to compute the solution of a component have been computed once we reach this component. To solve components, we use fixed-point iteration and Newton's method. Exploring the use of other solution methods is a potential venue for future performance improvements.

In sum, the method described here involves creating an intermediate representation, a FSPN, from a probabilistic program, which can then be efficiently solved to compute the required distribution. This method often makes nested query models useful in practice: as mentioned before, we have used it for all of the examples in Chapter 4.

5.4.3 Technical ingredients

Algorithm 5-1 shows `BuildFSPN`, the algorithm described in the previous section. This procedure takes as arguments an interpreter in factored coroutine form and an initial interpreter argument x_{init}. The algorithm steps through all possible execution paths while building the corresponding factored sum-product network, and avoiding duplicate evaluation of subproblems. We now describe three ingredients for this procedure—the task queue, constant-time subcall identification, and factorization grain.

Task queue. In programs with self-recursive calls, the exploration order of different execution paths can be highly constrained. For example, in order to evaluate the first `if`-branch of `(game true)` in Figure 5-1, we need to know at least one of the return values of `(game (not true))`, but these in turn depend on the return values of `(game true)`. In order to let program exploration be guided by what return values are known, we maintain a map `terminals`, which maps each root node to all known terminal values reachable from it, and a map `callbacks`, which maps each root node to a list of callbacks. A callback is a pair of a node n and a continuation c. When a new terminal v is found below a root node associated with callback (n, c), the call $c(v)$ is added to the task queue and used to continue evaluation and network building in the original context.

Constant-time subcall identification. At each subcall, we need to determine whether the subcall is new or whether it has already been assigned a FSPN node. For the algorithm to have constant-time overhead over steps of the underlying interpreter, it is crucial that this computation takes place in constant time, i.e., it must not depend on the size of the interpreter arguments. This suggests the use of an underlying interpreter that represents values in a compressed way, e.g., using the

value-number technique described in Aho et al. [2]. In Algorithm 5-1, this happens in `subproblem`, a map from interpreter arguments to network nodes.

Factorization grain. There are two ways to determine how much information we share between subcalls: (1) While the interpreter may cede control at all recursive calls, it does not need to for our algorithm to be valid. There is a continuum between building a fully factored FSPN and building a tree of random choices without factorization. In our experiments, we have found it advantageous to factor at all calls that correspond to applications of top-level functions. (2) What information the underlying interpreter passes to its recursive calls affects sharing. In Church, where interpreter arguments consist of expressions and environments, restricting environments to *relevant* environments is critical for efficient Dynamic Programming.

5.5 Empirical evaluation

In this section, we describe three situations where we have found generic Dynamic Programming to be useful: teaching probabilistic models, research in computational cognitive science, and analysis of multi-agent reasoning in game-theoretic situations. We present an example for each of these situations and compare Dynamic Programming to other inference algorithms.

In **teaching probabilistic models**, we usually aim to present modeling and inference patterns before we discuss the internals of inference algorithms, since the former provide motivation for the latter. The *Probabilistic Models of Cognition* tutorial by Goodman et al. [28] follows this approach and has been used in graduate classes at MIT and Stanford. Since implementations of probabilistic programming languages supply universal inference algorithms, it is possible to allow students to experiment with models and solve exercises without requiring deep understanding of

Algorithm 5-1: Compiling probabilistic programs to factored sum-product networks

```
procedure BuildFSPN(I, x_init)
    G = Graph()
    r = G.addNode(root)
    Q = [(λ.I(x_init), r, 1.0)]
    terminals, callbacks, subproblem = {}, {}, {}
    while Q is not empty do
        (f, n_prev, w_prev) = Q.pop()
        x = f()
        if x is a value v then
            n_cur = G.addNode(indicator, v)
            r = G.root[n_prev]
            if v ∉ terminals[r] then
                for all (n', c) in callbacks[r] do
                    processTerminal(G, Q, r, v, n', c)
                end for
                terminals[r].add(v)
            end if
        else if x is a random choice (c, v⃗, p⃗) then
            n_cur = G.addNode(sum)
            for all v, p ∈ v⃗, p⃗ do
                Q.enqueue(λ.c(v), n_cur, p)
            end for
        else if x is a subcall (c, s) then
            n_cur = G.addNode(sum)
            if s ∉ subproblem.keys() then
                r = G.addNode(root)
                subproblem[s] = r
                Q.enqueue(λ.I(s), r, 1.0)
            else
                r = subproblem[s]
                for all v ∈ terminals[r] do
                    processTerminal(G, Q, r, v, n_cur, c)
                end for
            end if
            callbacks[r].add((n_cur, c))
        end if
        G.addEdge(n_prev, n_cur, w_prev)
    end whilereturn G
end procedure

procedure ProcessTerminal(G, Q, n_root, v, n_prev, c)
    n_prod = G.addNode(product)
    n_ref = G.addNode(ref, n_root, v)
    G.addEdge(n_prev, n_prod, 1.0)
    G.addEdge(n_prod, n_ref, 1.0)
    Q.enqueue(λ.c(v), n_prod, 1.0)
end procedure
```

the underlying inference machinery.

However, the performance of existing "universal" algorithms strongly depends on the structure of the models they are applied to, even for models with a very small number of variables. Rejection sampling is only feasible as long as we do not condition on low-probability events; MCMC requires that the distribution does not have modes that are isolated with respect to the proposal structure of the algorithm.

Even in cases where sampling is feasible, it poses a challenge to students: it can be difficult to distinguish approximation noise from systematic inference patterns. For Metropolis-Hastings in the space of program traces [26, 101], no quantitative analysis of mixing times exists so far, hence analysis of convergence can be difficult even for experts; students' lack of background knowledge exacerbates this effect.

For example, consider the *rope-pulling game* (Figure 5-4), a simple probabilistic program without nested conditioning. Figure 5-5 shows how the L1 error between the estimated and true posterior distribution develops over time for rejection, MCMC, and Dynamic Programming. While MCMC has difficulty mixing between modes, and while rejection computes estimates using very few samples due to a low-probability condition, Dynamic Programming deterministically returns the exact answer after about 6 seconds.

In **cognitive science research**, we wish to quickly explore a wide range of model variations. While the model prototypes used in research have a tiny number of variables compared to the state of the art in machine learning, they are structurally complex and use features such as mutual recursion, nested conditioning, and stochastic higher-order functions. Probabilistic programming makes it possible to explore this space without building custom inference algorithms.

The caveat that the performance of current sampling-based algorithms strongly depends on models, even in small state spaces, applies here as well. For example,

```
(query

  ;; Generative model
  (define team1 (list 0 1))
  (define team2 (list 2 3))
  (define strengths
    (repeat 4 (λ () (if (flip) 10 5))))
  (define (strength person)
    (list-ref strengths person))
  (define (lazy person)
    (flip (/ 1 3)))
  (define (total-pulling team)
    (sum
     (map (λ (person)
            (if (lazy person)
                (/ (strength person) 2)
                (strength person)))
          team)))
  (define (winner team1 team2)
    (if (< (total-pulling team1)
           (total-pulling team2))
           'team2
           'team1))

  ;; Query expression
  (list (strength 0) (strength 1))

  ;; Condition
  (and
    (eq? 'team1 (winner team1 team2))
    ...
    (eq? 'team2 (winner team1 team2))))
```

Figure 5-4: The rope-pulling game, a simple generative model used in teaching probabilistic modeling.

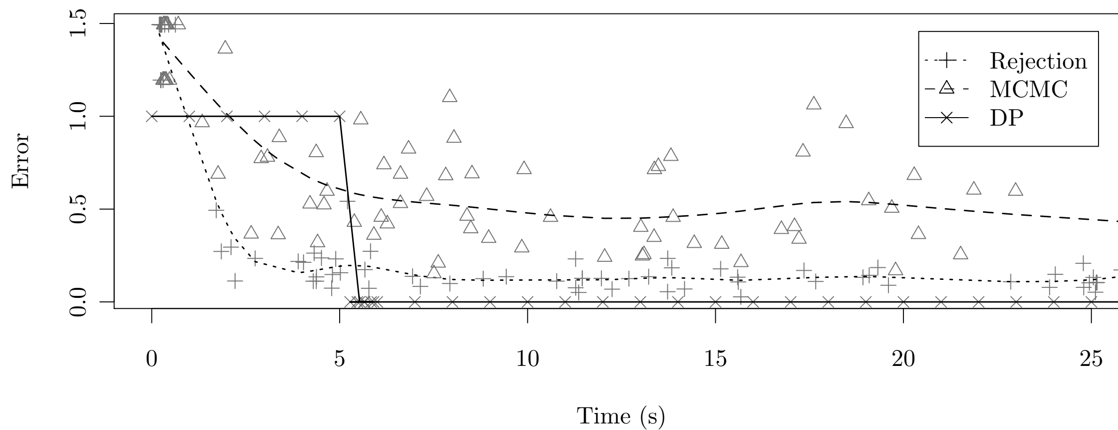


Figure 5-5: **Convergence to true distribution for the rope-pulling model.** Each point represents the L1 error between the estimated and true distribution for a given runtime and algorithm. While all algorithms eventually converge to the correct distribution for this model used in teaching, the only algorithm that quickly provides a *precise* answer is Dynamic Programming. In this example, MCMC has difficulty mixing between modes. Rejection computes estimates using very few samples due to a low-probability condition.

in Chapter 4 we proposed a model of language understanding based on the idea that listeners assume that speakers choose their utterances approximately optimally, and that listeners interpret an utterance by using Bayesian inference to "invert" this model of the speaker. Figure 4-3 shows part of a model of this type that predicts an interaction between the speaker's state of knowledge and the listener's interpretation of scalar implicatures (e.g., "some" implies "not all"). Using Dynamic Programming, the time it takes to compute the marginal distribution for this model grows linearly in the depth of recursive reasoning (Figure 5-6), whereas for current sampling techniques, inference time grows exponentially.

Moreover, some of the model features that are of research interest do not easily fit into the sampling framework. For example, in softmax-optimal decision-making, an action a is chosen according to exponentiated expected utility under a belief distribution $P(s)$, i.e., $P(a) \propto \exp\left(\alpha \mathbb{E}_{P(s)}[U(a;s)]\right)$. A direct translation into a probabilistic language with sampling semantics seems to require additional programming constructs that reify distributions. Such constructs can be provided more easily in the setting of exact inference.

The analysis of **multi-agent reasoning in game-theoretic situations** shares many properties with cognitive science research, but places even more emphasis on multiply nested conditioning. This commonly rules out existing sampling-based algorithms. At the same time, enumeration is often not an option either, since exploiting shared structure is critical in reducing the state space to tractable size: in the analysis of multiple agents thinking about one another, we can share computation between all agents, actual and counterfactual, that are modeled as being in the same state of mind.

As a particularly difficult example, consider the *Blue-Eyed Islanders* puzzle, a well-known problem in epistemic logic [90] that we already encountered in Section

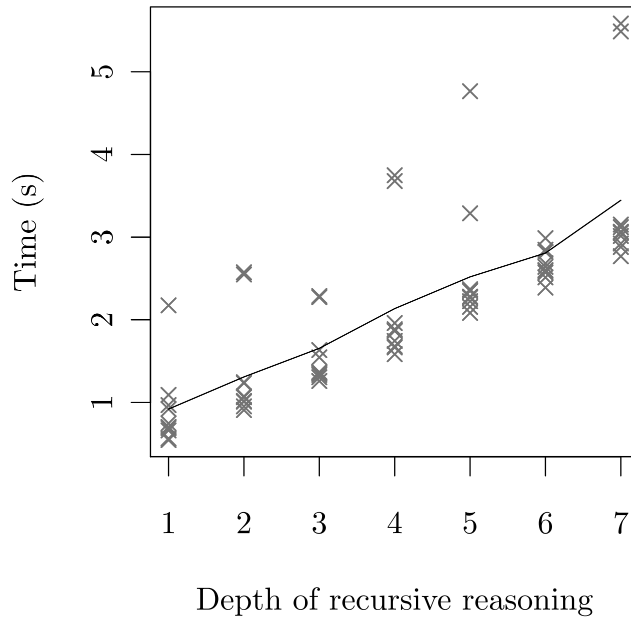


Figure 5-6: **Increase in Dynamic Programming inference time as a function of nested conditioning depth.** For the language understanding model shown in Figure 4-3, Dynamic Programming makes it possible to explore nested recursive conditioning with linear growth of inference time in the depth of recursion. Each point on the plot corresponds to a run of our algorithm on the model with a given depth. For rejection and MCMC over rejection, expected inference time grows exponentially.

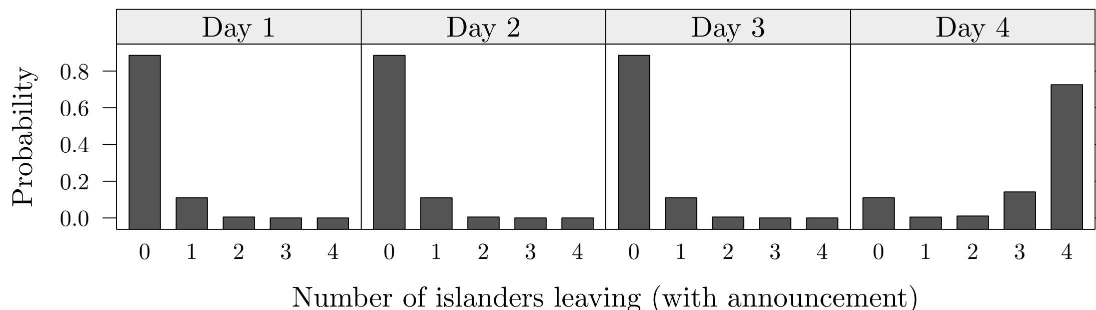


Figure 5-7: While the Blue-Eyed Islanders puzzle is challenging for all generic inference algorithms, Dynamic Programming allows predictions for small population sizes that are already intractable for MCMC and rejection. The figure shows results for population size 4, all islanders blue-eyed: on the 4th day, it is highly likely that all islanders decide to leave.

4.6. Results for a stochastic version of this puzzle are shown in Figure 5-7. The difficulty of this model stems from the fact that each day, every islander reasons about the reasoning of all of the other islanders on the previous day, and that their reasoning must again include all islanders' reasoning on the day before the previous day, etc. However, due to the symmetry of the setup, all islanders with blue eyes and all islanders without blue eyes do the same computation on any given day. Their computations are merged by our algorithm, which makes exact inference feasible for small populations.

5.6 Related work

The Dynamic Programming algorithm presented here is related to a wide range of algorithms which use Dynamic Programming to reuse computation in natural language processing, logic programming, and functional programming. For instance, Klein and Manning [43] exploit strongly connected components to perform efficient

exact marginalization for PCFGs in a way similar to the present algorithm. It is also known that, in general, exactly solving tasks such as marginalization for recursive programs leads to systems of nonlinear equations (see, e.g., comments in [13]).

A review of the connections to the many individual algorithms in the literature is beyond the scope of this work. Instead, we focus on three systems that provide generic, exact inference algorithms for universal probabilistic (or weighted) modeling languages: IBAL, PRISM, and Dyna.

The functional programming language IBAL [65] is the system most closely related to the present work. IBAL is a probabilistic variant of ML that provides an exact marginalization algorithm for discrete probabilistic models, which is based on a generalization of variable elimination. The graph used by this algorithm also exploits sharable subcomputations across the evaluation of the probabilistic program. However, the present algorithm is more general than the IBAL algorithm in the following sense: The IBAL algorithm relies on *acyclic* computation graphs; this is equivalent to the requirement that the computation be *evidence-finite* [46]—there must only be a finite number of computations which can give rise to the observed evidence. By contrast, our algorithm handles many cases of *evidence-infinite* computation. For example, the simple recursive program shown in Figure 5-1, which has finite support {`true`, `false`} but an infinite number of computations which give rise to each support value, cannot be marginalized by IBAL, but is correctly handled by our algorithm. Practical examples of such evidence-infinite computations include the nested-query models for multi-agent reasoning that we have described above.

PRISM [72] is another system similar in spirit to the present work. PRISM is a probabilistic generalization of Prolog, which provides a generic inference algorithm based on Dynamic Programming. PRISM is able to recover many standard algorithms (e.g., the inside-outside algorithm for PCFG estimation), but like IBAL, it

cannot handle evidence-infinite computations.

Dyna [13] takes a somewhat different approach. Dyna is a programming language for expressing weighted deductive logic programs. Dyna makes use of generalizations of parsing-as-deduction [79] and semi-ring parsing [22] to compile weighted logic programs into highly optimized Dynamic Programs. Dyna differs from our algorithm in the target level of abstraction. Our algorithm is focused on the problem of rapid prototyping of models for which no standard Dynamic Programming algorithm exists. The programmer simply provides an interpreter, and our algorithm automatically exploits whatever sharing is exposed by the structure of the recursive calls made in the process of computing the marginal distribution for a particular model. By contrast, Dyna is a language for abstractly expressing *specific* Dynamic Programming *algorithms* and compiling these algorithms to highly efficient code. It allows the programmer lower-level control over algorithm specification, but it also *requires* the programmer to specify these algorithmic details.

5.7 Conclusion

We have developed a Dynamic Programming algorithm for exact inference in probabilistic programs. We have illustrated how this algorithm aids the use of probabilistic programs in teaching and research. Future work includes incorporating techniques from other approaches to Dynamic Programming (such as evidence propagation from IBAL, and efficient code generation from Dyna) and exploring techniques for approximate Dynamic Programming.

Exact inference will probably not scale to models with realistic state spaces. This poses the practical question of finding approximate inference algorithms that work in the setting of nested queries, and the scientific question of understanding how

humans cope with the same. For some versions of these problems, scalable algorithms have already been introduced; for example, Monte Carlo tree search for playing the board game Go [44]. It may be possible to construct inference algorithms for probabilistic programs that reduce to such known algorithms in restricted settings. Conversely, studying the psychological process of inference—which may implicate parallel processing, simulation, and significant resource bounds—may suggest new algorithms to solve the engineering challenges of inference.

Chapter 6

Learning Stochastic Inverses

6.1 Introduction

Bayesian inference is computationally expensive. Even approximate, sampling-based algorithms tend to take many iterations before they produce reasonable answers. In contrast, human recognition of words, objects, and scenes is extremely rapid, often taking only a few hundred milliseconds—only enough time for a single pass from perceptual evidence to deeper interpretation. Yet human perception and cognition are often well-described by probabilistic inference in complex models. How can we reconcile the speed of recognition with the expense of coherent probabilistic inference? How can we build systems, for applications like robotics and medical diagnosis, that exhibit similarly rapid performance at challenging inference tasks?

One response to such questions is that these problems are not, and should not be, solved from scratch each time they are encountered. Humans and robots are in the setting of *amortized inference*: they have to solve many similar inference problems,

---

This chapter is based on Stuhlmüller et al. [85].

and can thus offload part of the computational work to shared precomputation and adaptation over time. This raises the question of which kinds of precomputation and adaptation are useful. There is substantial previous work on adaptive inference algorithms, including Cheng and Druzdzel [11], Haario et al. [32], Ortiz and Kaelbling [63], Roberts and Rosenthal [69]. While much of this work is focused on adaptation for a single posterior inference, amortized inference calls for adaptation across many different inferences. In this setting, we will often have considerable training data available in the form of posterior samples from previous inferences; how should we use this data to adapt our inference procedure?

We consider using training samples to learn the *inverse* structure of a directed model. Posterior inference is the task of inverting a probabilistic model: Bayes' theorem turns $p(d|h)$ into $p(h|d)$; vision is commonly understood as inverse graphics [37] and, more recently, as inverse physics [71, 95]; and conditional inference in probabilistic programs can be described as "running a program backwards" [e.g., 99]. However, while this is a good description of the problem that inference solves, conditional sampling usually does *not* proceed backwards step-by-step. We suggest taking this view more literally and actually learning the inverse conditionals needed to invert the model. For example, consider the Bayesian network shown in Figure 6-1. In addition to the default "forward" factorization shown on the left, we can consider an "inverse" factorization shown on the right. Knowing the conditionals for this inverse factorization would allow us to rapidly sample the latent variables given an observation. In this chapter, we will explore what these factorizations look like for Bayesian networks, how to learn them, and how to use them to construct block proposals for MCMC.

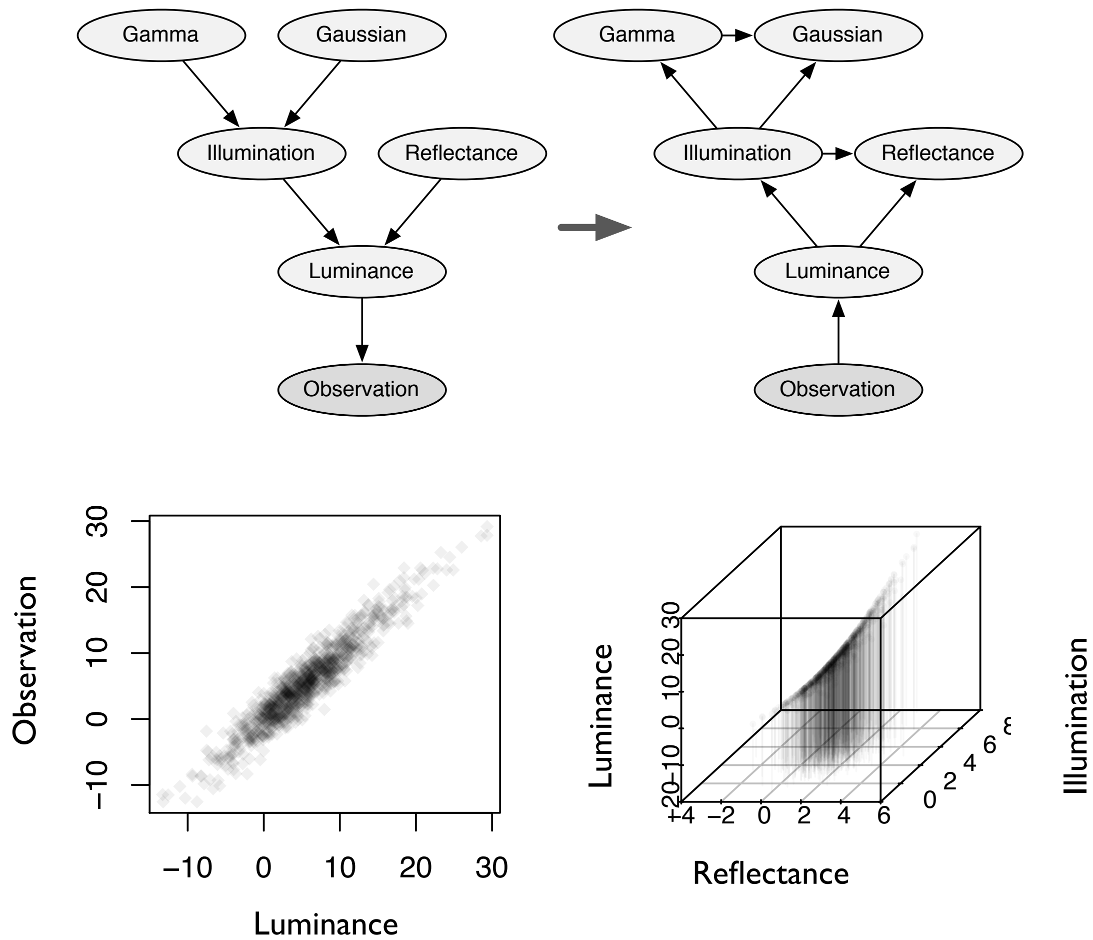


Figure 6-1: A Bayesian network modeling brightness constancy in visual perception, a possible inverse factorization, and two of the local joint distributions that determine the inverse conditionals.

6.2 Inverse factorizations

Let p be a distribution on latent variables $x = (x_1, \ldots, x_m)$ and observed variables $y = (y_1, \ldots, y_n)$. A Bayesian network G is a directed acyclic graph that expresses a factorization of this joint distribution in terms of the distribution of each node conditioned on its parents in the graph:

$$p(x, y) = \prod_{i=1}^{m} p(x_i|\mathrm{pa}_G(x_i)) \prod_{j=1}^{n} p(y_j|\mathrm{pa}_G(y_j))$$

When interpreted as a generative (causal) model, the observations y typically depend on a non-empty set of parents, but are not themselves parents of any nodes.

In general, a distribution can be represented using many different factorizations. We say that a Bayesian network H expresses an *inverse factorization* of p if the observations y do not have parents (but may themselves be parents of some x_i):

$$p(x, y) = p(y) \prod_{i=1}^{m} p(x_i|\mathrm{pa}_H(x_i))$$

As an example, consider the forward and inverse networks shown in Figure 6-1. We call the conditional distributions $p(x_i|\mathrm{pa}_H(x_i))$ *stochastic inverses*, with *inputs* $\mathrm{pa}_H(x_i)$ and *output* x_i. If we could sample from these distributions, we could produce samples from $p(x|y)$ for arbitrary y, which solves the problem of inference for all queries with the same set of observation nodes.

In general, there are many possible inverse factorizations. For each latent node, we can find a factorization such that this node does not have children. This fact will be important in Section 6.4 when we resample subsets of inverse graphs. Algorithm 6-1 gives a heuristic method for computing an inverse factorization given Bayes net

Algorithm 6-1: Heuristic inverse factorization

Input: Bayesian network G with latent nodes x and observed nodes y; desired leaf node x_i

Output: Ordered inverse graph H

1: order x such that nodes close to y are first, leaf node x_i is last
2: initialize H to empty graph
3: add nodes y to H
4: **for** node x_j **in** x **do**
5: add x_j to H
6: set $\text{pa}_H(x_j)$ to a minimal set of nodes in H that d-separates x_j from the remainder of H based on the graph structure of G
7: **end for**

G, observation nodes y, and desired leaf node x_i. We compute an ordering on the nodes of the original Bayes net from observations to leaf node. We then add the nodes in order to the inverse graph, with dependencies determined by the graph structure of the original network.

In the setting of amortized inference, past tasks provide approximate posterior samples for the corresponding observations. We therefore investigate learning inverses from such samples, and ways of using approximate stochastic inverses for improving the efficiency of solving future inference tasks.

6.3 Learning stochastic inverses

It is easy to see that we can estimate conditional distributions $p(x_i|\text{pa}_H(x_i))$ using samples S drawn from the prior $p(x, y)$. For simplicity, consider discrete variables and an empirical frequency estimator:

$$\theta_S(x_i|\text{pa}_H(x_i)) = \frac{|\{s \in S : x_i^{(s)} \wedge \text{pa}_H^{(s)}(x_i)\}|}{|\{s \in S : \text{pa}_H^{(s)}(x_i)|}$$

Because θ_S is a consistent estimator of the probability of each outcome for each setting of the parent variables, the following theorem follows immediately from the strong law of large numbers:

Theorem 1. (Learning from prior samples) *Let H be an inverse factorization. For samples S drawn from $p(x, y)$, $\theta_S(x_i|\text{pa}_H(x_i)) \to p(x_i|\text{pa}_H(x_i))$ almost surely as $|S| \to \infty$.*

Samples generated from the prior may be sparse in regions that have high probability under the posterior, resulting in slow convergence of the inverses. We now show that valid inverse factorizations allow us to learn from posterior samples as well.

Theorem 2. (Learning from posterior samples) *Let H be an inverse factorization. For samples S drawn from $p(x|y)$, $\theta(x_i|\text{pa}_H(x_i)) \to p(x_i|\text{pa}_H(x_i))$ almost surely as $|S| \to \infty$ for values of $\text{pa}_H(x_i)$ that have positive probability under $p(x|y)$.*

Proof. For values $\text{pa}_H(x_i)$ that are not in the support of $p(x|y)$, $\theta(x_i|\text{pa}_H(x_i))$ is undefined. For values $\text{pa}_H(x_i)$ in the support, $\theta(x_i|\text{pa}_H(x_i)) \to p(x_i|\text{pa}_H(x_i), y)$ almost surely. By definition, any node in a Bayesian network is independent of its non-descendants given its parent variables. The nodes y are root nodes in H and hence do not descend from x_i. Therefore, $p(x_i|\text{pa}_H(x_i), y) = p(x_i|\text{pa}_H(x_i))$ and the theorem holds. □

Theorem 2 implies that we can use posterior samples from one observation set to learn inverses that apply to all other observation sets—while samples from $p(x|y)$ only provide global estimates for the given posterior, it is guaranteed that the local estimates created by the procedure above are equivalent to the query-independent conditionals $p(x_i|\text{pa}_H(x_i))$. In addition, we can combine samples from distributions

Algorithm 6-2: K-nearest neighbor density predictor

Input: Variable index i, inverse inputs z, samples S, number of neighbors k
Output: Sampled value for node x_i

1: retrieve k nearest pairs $(z^{(1)}, x_i^{(1)}), \ldots, (z^{(k)}, x_i^{(k)})$ in S based on distance to z
2: construct density estimate q on $x_i^{(1)}, \ldots, x_i^{(k)}$
3: sample from q

conditioned on several different observation sets to produce more accurate estimates of the inverse conditionals.

In the discussion above, we can replace θ with any consistent estimator of $p(x_i|\mathrm{pa}_H(x_i))$. We can also trade consistency for faster learning and generalization. This framework can make use of any supervised machine learning technique that supports sampling from a distribution on predicted outputs. For example, for discrete variables we can employ logistic regression, which provides fast generalization and efficient sampling, but cannot, in general, represent the posterior exactly. Our choice of predictor can be data-dependent—for example, we can add interaction terms to a logistic regression predictor as more data becomes available.

For continuous variables, consider a predictor based on k-nearest neighbors that produces samples as follows (Algorithm 6-2): Given new input values z, retrieve the k previously observed input-output pairs that are closest to the current input values. Then, use a consistent density estimator to construct a density estimate on the nearby previous outputs and sample an output x_i from the estimated distribution.

Showing that this estimator converges to the true conditional density $p(x|z)$ is more subtle. If the conditional densities are smooth in the sense that:

$$\forall \varepsilon > 0 \; \exists \delta > 0 : \forall z_1, z_2 \; d(z_1, z_2) < \delta \Rightarrow D_{\mathrm{KL}}(p(x|z_1), p(x|z_2)) < \varepsilon$$

then we can achieve any desired accuracy of approximation by assuring that the nearest neighbors used all lie within a δ-ball, but that the number of neighbors goes to infinity. We can achieve this by increasing k slowly enough in $|S|$. The exact rate at which we may increase k depends on the distribution and may be difficult to determine.

6.4 Inverse MCMC

We have described how to compute the structure of inverse Bayes nets, and how to learn the associated conditional distributions and densities from prior and posterior samples. This produces fast, but possibly biased recognition models. To get a consistent estimator, we use these recognition models as part of a Metropolis-Hastings scheme that, as the amount of training data grows, converges to Gibbs sampling for proposals of size 1, to blocked-Gibbs for larger proposals, and to perfect posterior sampling for proposals of size $|G|$.

We propose the following *Inverse MCMC* procedure (Algorithm 6-3): *Offline*, use Algorithm 6-1 to compute an inverse graph for each latent node and train each local inverse in this graph from (posterior or prior) samples. *Online*, run Metropolis-Hastings with the proposal mechanism shown in Algorithm 6-4, which resamples a set of up to k variables using the trained inverses[1]. With little training data, we will want to make small proposals (small k) in order to achieve a reasonable acceptance rate; with more training data, we can make larger proposals and expect to succeed.

Theorem 3. *Let G be a Bayesian network, let θ be a consistent estimator (for inverse conditionals), let $\{H_i\}_{i\in 1..m}$ be a collection of inverse graphs produced using*

[1] In a setting where we only ever resample up to k variables, we only need to estimate the relevant inverses, i.e., not all conditionals for the full inverse graph.

Algorithm 6-1, and assume a source of training samples (prior or posterior) with full support. Then, as training set size $|S| \to \infty$, Inverse MCMC with proposal size k converges to block-Gibbs sampling where blocks are the last k nodes in each H_i. In particular, it converges to Gibbs sampling for proposal size $k = 1$ and to exact posterior sampling for $k = |G|$.

Proof. We must show that proposals are made from the conditional posterior in the limit of large training data. Fix an inverse H, and let $\mathbf{x}$ be the last k variables in H. Let $\text{pa}_H(\mathbf{x})$ be the union of H-parents of variables in $\mathbf{x}$ that are not themselves in $\mathbf{x}$. By construction according to Algorithm 6-1, $\text{pa}_H(\mathbf{x})$ form a Markov blanket of $\mathbf{x}$ (that is, $\mathbf{x}$ is conditionally independent of other variables in G, given $\text{pa}_H(\mathbf{x})$). Now the conditional distribution over $\mathbf{x}$ factorizes along the inverse graph: $p(\mathbf{x}|\text{pa}_H(\mathbf{x})) = \prod_{i=k}^{|H|} p(x_i|\text{pa}_H(x_i))$. But by Theorems 1 and 2, the estimators θ converge, when they are defined, to the corresponding conditional distributions, $\theta(x_i|\text{pa}_H(x_i)) \to p(x_i|\text{pa}_H(x_i))$; since we assume full support, $\theta(x_i|\text{pa}_H(x_i))$ is defined wherever $p(x_i|\text{pa}_H(x_i))$ is defined. Hence, using the estimated inverses to sequentially sample the $\mathbf{x}$ variables results, in the limit, in samples from the conditional distribution given remaining variables. (Note that, in the limit, these proposals will always be accepted.) This is the definition of block-Gibbs sampling. The special cases of $k = 1$ (Gibbs) and $k = |G|$ (posterior sampling) follow immediately. □

Instead of learning the k=1 "Gibbs" conditionals for each inverse graph, we can often precompute these distributions to "seed" our sampler. This suggests a bootstrapping procedure for amortized inference on observations $y^{(1)}, \ldots, y^{(t)}$: first, precompute the "Gibbs" distributions so that k=1 proposals will be reasonably effective; then iterate between training on previously generated approximate posterior samples and doing inference on the next observation. Over time, increase the size of proposals,

Algorithm 6-3: Inverse MCMC

Input: Prior or posterior samples S
Output: Samples $x^{(1)}, \ldots, x^{(T)}$

Offline (train inverses):

1: **for** i **in** $1 \ldots m$ **do**
2: $\quad H_i \leftarrow$ from Algorithm 6-1
3: $\quad$ **for** j **in** $1 \ldots m$ **do**
4: $\quad\quad$ train inverse $\theta_S(x_j|\text{pa}_{H_i}(x_j))$
5: $\quad$ **end for**
6: **end for**

Online (MH with inverse proposals):

1: **for** t **in** $1 \ldots T$ **do**
2: $\quad x'$, p_{fw}, p_{bw} from Algorithm 6-4
3: $\quad x \leftarrow x'$ with MH acceptance rule
4: **end for**

Algorithm 6-4: Inverse MCMC proposer

Input: State x, observations y, ordered inverse graphs $\{H_i\}_{i \in 1..m}$, proposal size $k_{\max}$, inverses θ
Output: Proposed state x', forward and backward probabilities p_{fw} and p_{bw}

1: $H \sim \text{Uniform}(\{H_i\}_{i \in 1..m})$
2: $k \sim \text{Uniform}(\{0, 1, \ldots, k_{\max} - 1\})$
3: $x' \leftarrow x$
4: $p_{\text{fw}}, p_{\text{bw}} \leftarrow 0$
5: **for** j **in** $n - k, \ldots, n$ **do**
6: $\quad$ let x_l be jth variable in H
7: $\quad x'_l \sim \theta(x_l|\text{pa}_H(x'_l))$
8: $\quad p_{\text{fw}} \leftarrow p_{\text{fw}} \cdot p_\theta(x'_l|\text{pa}_H(x'_l))$
9: $\quad p_{\text{bw}} \leftarrow p_{\text{bw}} \cdot p_\theta(x_l|\text{pa}_H(x_l))$
10: **end for**

possibly depending on acceptance ratio or other heuristics.

For networks with nearly-deterministic dependencies, Gibbs may be unable to generate training samples of sufficient quality. This poses a chicken-and-egg problem: we need a sufficiently good posterior sampler to generate the data required to train our sampler. To address this problem, we propose a simple annealing scheme: We introduce a temperature parameter t that controls the extent to which (almost-)deterministic dependencies in a network are relaxed. We produce a sequence of trained samplers, one for each temperature, by generating samples for a network with temperature t_{i+1} using a sampler trained on approximate samples for the network with next-higher temperature t_i. Finally, we discard all samplers except for the sampler trained on the network with $t = 0$, the network of interest.

In the next section, we explore the practicality of such bootstrapping schemes as well as the general approach of Inverse MCMC.

6.5 Experiments

We are interested in networks such that (1) there are many layers of nodes, with some nodes far removed from the evidence, (2) there are many observation nodes, allowing for a variety of queries, and (3) there are strong dependencies, making local Gibbs moves challenging.

We start by studying the behavior of the Inverse MCMC algorithm with empirical frequency estimator on a 225-node rectangular grid network from the UAI 2008 inference competition. This network has binary nodes and approximately 50% deterministic dependencies, which we relax to dependencies with strength .99. We select the 15 nodes on the diagonal as observations and remove any nodes below, leaving a triangular network with 120 nodes and treewidth 15 (Figure 6-2). We compute the true marginals P^* using IJGP [55], and calculate the error of our estimates P^s as

$$\text{error} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{|X_i|} \sum_{x_i \in X_i} |P^*(X_i = x_i) - P^s(X_i = x_i)|.$$

We generate 20 inference tasks as sources of training samples by sampling values for the 15 observation nodes uniformly at random. We precompute the "final" inverse conditionals as outlined above, producing a Gibbs sampler when k=1. For each inference task, we use this sampler to generate 10^5 approximate posterior samples.

Figures 6-3 and 6-5 show the effect of training the frequency estimator on 10 inference tasks and testing on a different task (averaged over 20 runs). Inverse proposals of (up to) size k=20 do worse than pure Gibbs sampling with little training (due to higher rejection rate), but they speed convergence as the number of training samples increases. More generally, large proposals are likely to be rejected without training, but improve convergence after training.

Figure 6-6 illustrates how the number of inference tasks influences error and MH acceptance ratio in a setting where the total number of training samples is kept constant. Surprisingly, increasing the number of training tasks from 5 to 15 has little effect on error and acceptance ratio for this network. That is, it seems relatively unimportant which posterior the training samples are drawn from; we may expect different results when posteriors are more sparse.

Figure 6-7 shows how different sources of training data affect the quality of the trained sampler (averaged over 20 runs). As the strength of near-deterministic dependencies increases, direct training on Gibbs samples becomes infeasible. In this regime, we can still train on prior samples and on Gibbs samples for networks with relaxed dependencies. Alternatively, we can employ the annealing scheme outlined in the previous section. In this example, we take the temperature ladder to be $[.2, .1, .05, .02, .01, 0]$—that is, we start by learning inverses for the relaxed network where all CPT probabilities are constrained to lie within $[.2, .8]$; we then use these inverses as proposers for MCMC inference on a network constrained to CPT probabilities in $[.1, .9]$, learn the corresponding inverses, and continue, until we reach the network of interest (at temperature 0).

While the empirical frequency estimator used in the above experiments provides an attractive asymptotic convergence guarantee (Theorem 3), it is likely to generalize slowly from small amounts of training data. For practical purposes, we may be more interested in getting useful generalizations quickly than converging to a perfect proposal distribution. Fortunately, the Inverse MCMC algorithm can be used with any estimator for local conditionals, consistent or not. We evaluate this idea on a 12-node subset of the network used in the previous experiments. We learn complete inverses, resampling up to 12 nodes at once. We compare inference using a logistic regression estimator with L_2 regularization (with and without interaction terms)

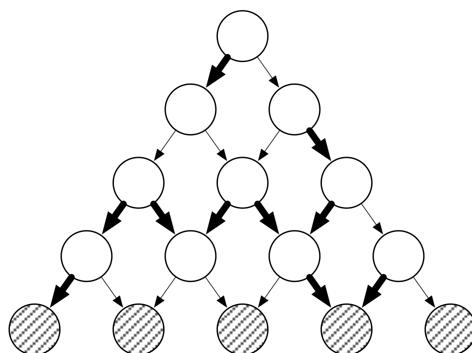


Figure 6-2: Schema of the Bayes net structure used in experiment 1. Thick arrows indicate almost-deterministic dependencies, shaded nodes are observed. The actual network has 15 layers with a total of 120 nodes.

to inference using the empirical frequency estimator. Figure 6-9 shows the error (integrated over time to better reflect convergence speed) against the number of training examples, averaged over 300 runs. The regression estimator with interaction terms results in significantly better results when training on few posterior samples, but is ultimately overtaken by the consistent empirical estimator.

Next, we use the KNN density predictor to learn inverse distributions for the continuous Bayesian network shown in Figure 6-1. To evaluate the quality of the learned distributions, we take 1000 samples using Inverse MCMC and compare marginals to a solution computed by JAGS [66]. As we refine the inverses using forward samples, the error in the estimated marginals decreases towards 0, providing evidence for convergence towards a posterior sampler (Figure 6-4).

To evaluate Inverse MCMC in more breadth, we run the algorithm on all binary Bayes nets with up to 500 nodes that have been submitted to the UAI 08 inference competition (216 networks). Since many of these networks exhibit strong determinism, we train on prior samples and apply the annealing scheme outlined above to generate approximate posterior samples. For training and testing, we use the evi-

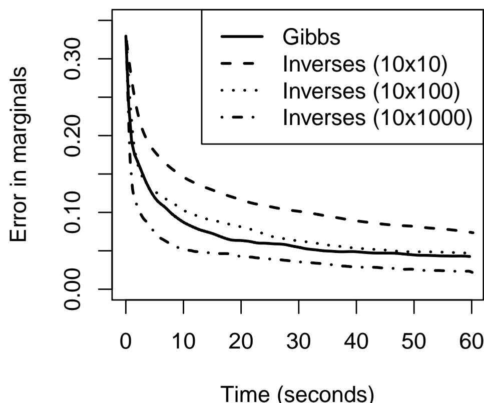


Figure 6-3: The effect of training on approximate posterior samples for 10 inference tasks. As the number of training samples per task increases, Inverse MCMC with proposals of size 20 performs new inference tasks more quickly.

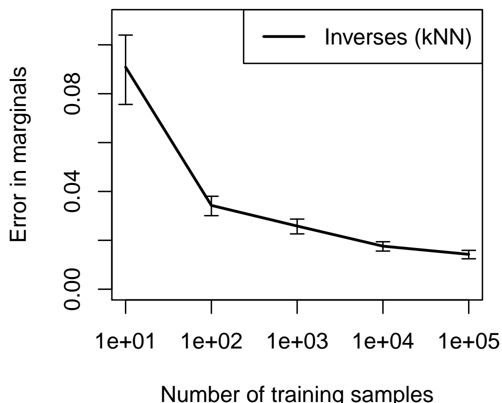


Figure 6-4: Learning an inverse distribution for the brightness constancy model (Figure 6-1) from prior samples using the KNN density predictor. More training samples result in better estimates after the same number of MCMC steps.

dence provided with each network. We compute the error in marginals as described above for both Gibbs (proposal size 1) and Inverse MCMC (maximum proposal size 20). To summarize convergence over the 1200s of test time, we compute the area under the error curves (Figure 6-8). Each point represents a single run on a single model. We label different classes of networks. For the grid networks, grid-k denotes a network with $k\%$ deterministic dependencies. While performance varies across network classes—with extremely deterministic networks making the acquisition of training data challenging—the comparison with Gibbs suggests that learned block proposals frequently help.

Overall, these results indicate that Inverse MCMC is of practical benefit for learn-

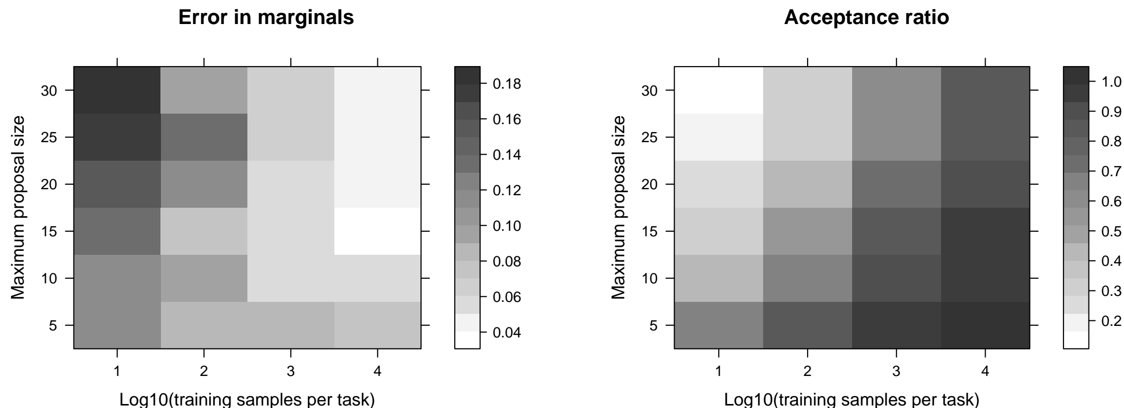


Figure 6-5: Without training, big inverse proposals result in high error, as they are unlikely to be accepted. As we increase the number of approximate posterior samples used to train the MCMC sampler, the acceptance probability for big proposals goes up, which decreases overall error.

ing block proposals in reasonably large Bayes nets and using a realistic amount of training data (an amount that might result from amortizing over five or ten inferences).

6.6 Related work

A recognition network [58] is a multilayer perceptron used to predict posterior marginals. In contrast to our work, a single global predictor is used instead of small, compositional prediction functions. By learning local inverses our technique generalizes in a more fine-grained way, and can be combined with MCMC to provide unbiased samples. Adaptive MCMC techniques such as those presented in Roberts and Rosenthal [69] and Haario et al. [32] are used to tune parameters of MCMC algorithms, but do not allow arbitrarily close adaptation of the underlying model to the posterior, whereas our method is designed to allow such close approxima-

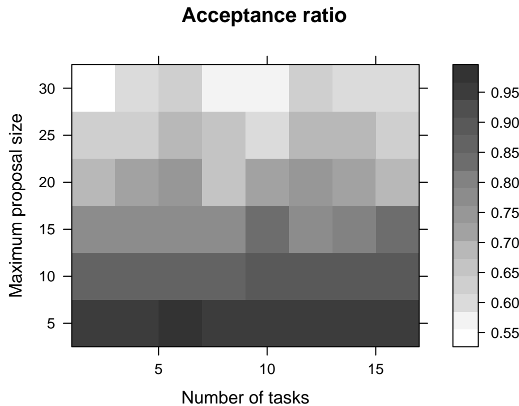


Figure 6-6: For the network under consideration, increasing the number of tasks we train on has little effect on acceptance ratio (and error) if we keep the total number of training samples constant.

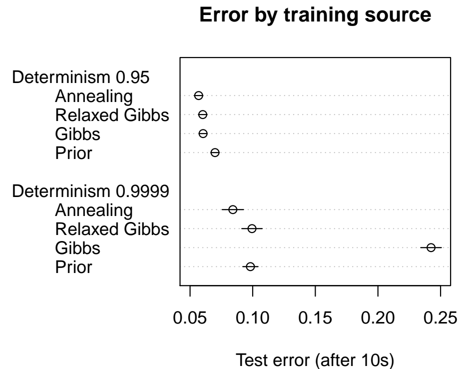
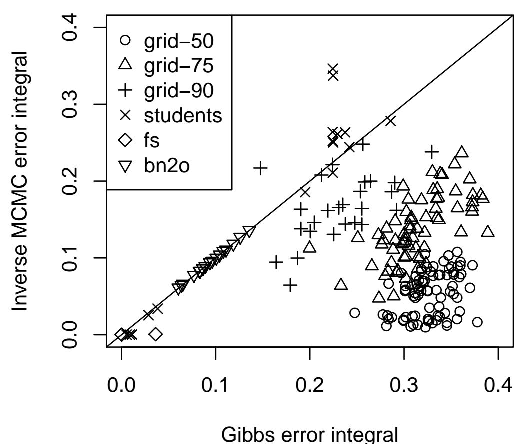


Figure 6-7: For networks without hard determinism, we can train on Gibbs samples. For others, we can use prior samples, Gibbs samples for relaxed networks, and samples from a sequence of annealed Inverse samplers.

tion. A number of adaptive importance sampling algorithms have been proposed for Bayesian networks, including Shachter and Peot [76], Cheng and Druzdzel [11], Yuan and Druzdzel [103], Yu and Van Engelen [102], Hernandez et al. [35], Salmeron et al. [70], and Ortiz and Kaelbling [63]. These techniques typically learn Bayes nets which are directed "forward", which means that the conditional distributions must be learned from posterior samples, creating a chicken-and-egg problem. Because our trained model is directed "backwards", we can learn from both prior and posterior samples. Gibbs sampling and single-site Metropolis-Hastings are known to converge slowly in the presence of determinism and long-range dependencies. It is well-known that this can be addressed using block proposals, but such proposals typically need to be built manually for each model. In our framework, block proposals are learned from past samples, with a natural parameter for adjusting the block size.



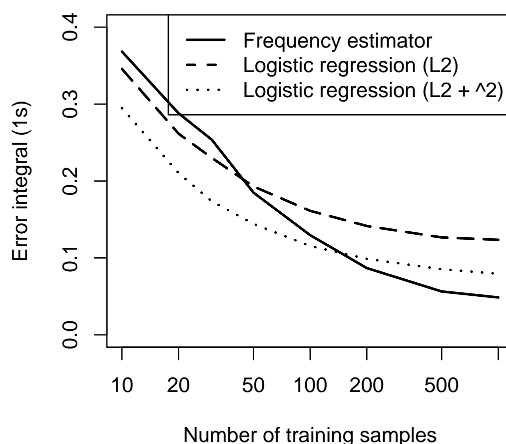


Figure 6-8: Each mark represents a single run of a model from the UAI 08 inference competition. Marks below the line indicate that integrated error over 1200s of inference is lower for Inverse MCMC than Gibbs sampling.

Figure 6-9: Integrated error (over 1s of inference) as a function of the number of samples used to train inverses, comparing logistic regression with and without interaction terms to an empirical frequency estimator.

6.7 Conclusion

We have described a class of algorithms for the setting of amortized inference, based on the idea of learning local stochastic inverses—the information necessary to "run a model backward". We have given simple methods for estimating and using these inverses as part of an MCMC algorithm. In exploratory experiments, we have shown how learning from past inference tasks can reduce the time required to estimate quantities of interest. Much remains to be done to explore this framework. Based on our results, one particularly promising avenue is to explore estimators that initially generalize quickly (such as regression), but back off to a sound estimator as the training data grows.

Chapter 7

Coarse-to-Fine Sequential Monte Carlo

7.1 Introduction

Imagine watching a tennis tournament. Your visual system makes fast and accurate inferences about the depth field (how far away are different patches?), the objects (is that a ball or racket?), their trajectories, and many other properties of the scene. A powerful intuition is that such feats of inference are enabled by *coarse-to-fine* reasoning: first getting a rough sense of where the action is in the scene, how far away it is, and so on; then refining this impression to pick out details. The appeal of coarse-to-fine reasoning is manifold. First, there is introspection: When faced with a complex reasoning task, it often helps to take a step back, try to understand the big picture, and then focus on what seems most promising. The big picture tends to have fewer moving parts, and its parts tend to be easier to understand.

This chapter is based on Stuhlmüller et al. [86].

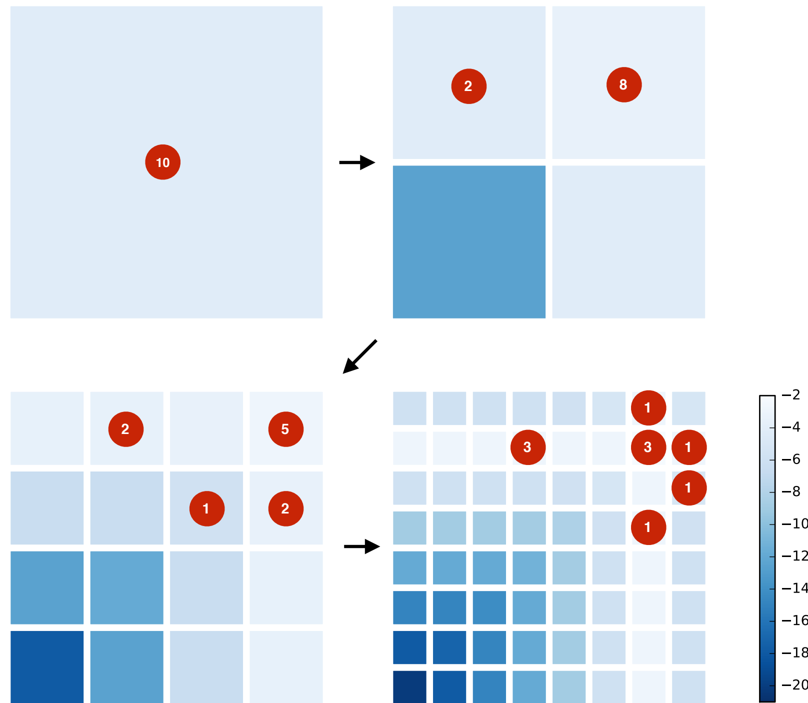


Figure 7-1: Incremental coarsening reduces surprise in SMC. Particles (red) are directed towards high-probability regions (light) step by step as we refine the state space from coarse to fine. The numbers indicate how many particles are associated with a particular state.

Neuroscience provides another angle: for instance, face processing in the high-level visual cortex plausibly follows coarse-to-fine principles [21], and stereoscopic depth perception similarly proceeds from large to small spatial scales [56]. Finally, there is a rich set of existing applications of coarse-to-fine techniques for specific applications in a diverse set of areas including physical chemistry [52], speech processing [88], PCFG parsing [10], and machine translation [64]. Despite the success and appeal of coarse-to-fine ideas, they have been difficult to apply in general settings. Here we propose a system for deriving coarse-to-fine inference from any model written as a probabilistic program. We do this by leveraging the program structure to transform

the initial program into a multi-level, coarse-to-fine program that can be used with existing inference algorithms.

Probabilistic programming languages provide a universal and high-level representation for probabilistic models, separating the burdens of modeling from those of inference. Yet the difficulty of inference can grow quickly as the state space (number of possible program executions) grows large. A widely-used technique for inference in large state spaces is Sequential Monte Carlo (SMC), a class of algorithms based on constructing a sequence of distributions, beginning with an easy-to-sample distribution and ending at the distribution of interest, with each distribution serving as an importance sampler for the next. The success of SMC rests on the quality of the approximating sequence. We present a generic method for deriving coarse-to-fine sequences of approximating distributions from a probabilistic program.

Our approach can be seen as building a hierarchical model from an initial model, where each stage of the hierarchy resolves more details of the state space than the one before. Additionally, we augment each level of the hierarchy with a coarse approximation to the evidence (implemented via *heuristic factors*), in order to specify a useful conditional distribution at each level. In practice we create the hierarchical model and heuristic factors simultaneously by specifying how to "lift" each element of the program—elementary distributions, primitive functions, and constants—to the coarser levels. The resulting model supports coarse-to-fine inference by SMC, where the nth distribution in the sequence is simply the state space of the n-coarsest levels; this implements inference for the original model correctly because, by construction, the marginal distribution over the finest level is the original distribution.

Model transformations let us directly use existing sequential inference algorithms to perform coarse-to-fine inference, rather than proposing a new inference algorithm *per se*. This is in contrast to essentially all prior work on coarse-to-fine inference,

including Kiddon and Domingos [42] and Steinhardt and Liang [80]. One benefit of this modular approach is that advances in SMC algorithms immediately yield improvements to coarse-to-fine inference. Another benefit is the conceptual clarity that comes from an explicit representation of the coarse-to-fine model.

In the following, we first review probabilistic programs and Sequential Monte Carlo. We then describe our coarse-to-fine program transform and how it lifts random variables, primitive functions, and factors to operate on multiple levels of abstraction. We apply this transform to three models in the domain of tracking partially observable objects over time given visual information: a depth-from-disparity model, a factorial hidden Markov model, and a model of visual scene understanding. We show that it can significantly reduce inference time in each of these domains. Finally, we discuss in what circumstances our proposed coarse-to-fine approach is a good fit, and outline research questions raised by this new approach to coarse-to-fine inference.

7.2 Background

7.2.1 Probabilistic programming in WebPPL

We express probabilistic programs in *WebPPL* [24], a small probabilistic language embedded in Javascript. This language is universal and feature-rich, so we expect the techniques to generalize to other languages. In this language, all random choices are marked by `sample`; the argument to `sample` is a distribution object (also called *Elementary Random Primitive*, or *ERP*), its return value a sample from this distribution. Calls to functions such as `flip(0.5)` are shorthand for `sample(bernoulliERP, [0.5])`.

To enable probabilistic conditioning, the language supports `factor` statements.

The argument to factor is a score: a number that is added to the log-probability of a program execution, thus increasing or decreasing its relative posterior probability. This includes hard conditioning as a special case (scores 0 and $-\infty$). Finally, the language supports inference primitives such as `ParticleFilter` and `MH` (Metropolis-Hastings). Each of these takes as an argument a thunk, that is, a stochastic function that itself takes no arguments. And each of these computes or estimates the distribution on return values of this thunk (its *marginal distribution*), taking into account the re-weighting induced by factor statements.

Figure 7-2a shows a program that implements a simplified one-step version of multiple object tracking: the noisily observed value 7 could have been produced by either x or y, each of which is uniformly chosen from $\{1, 2, \ldots, 8\}$. The final panel in Figure 7-1 shows the marginal distribution on $[x, y]$ for this program.

In probabilistic programs, the same syntactic variable can be used multiple times. The prototypical example is the geometric distribution:

```
var geometric = f() {
  return flip(0.1) ? 0 : 1 + geometric()
}
```

The call to `flip(0.1)` may occur an unbounded number of times. For many purposes, it is necessary to distinguish and refer to these different calls. In the context of MCMC, Wingate et al. [100] introduced a suitable naming scheme based on stack addresses. The address of a random choice is a list of syntactic locations, one for each function on the function call stack at the time when the random variable was sampled. We will build on this scheme to associate corresponding random choices on different levels of coarsening with each other, and use **address** in the following to refer to the current stack address.

```
var noisyObserve = f(obs){
  var score =
    -3*distance(obs, 7)
  factor(score)
}

var model = f(){
  var x = sample(uniformERP)
  var y = sample(uniformERP)
  var observation =
    flip(.5) ? x : y
  noisyObserve(observation)
  return [x, y]
}
```

(a) A probabilistic program

```
var noisyObserve = f(obs){
  var score =
    -3*distance(obs, 7)
  factor(score)
}

var model = f(){
  var x = sample(uniformERP)
  var heuristicScore =
    -distance(x, 7)
  factor(heuristicScore)
  var y = sample(uniformERP)
  var observation =
    flip(.5) ? x : y
  noisyObserve(observation)
  factor(-heuristicScore)
  return [x, y]
}
```

(b) Rewritten using heuristic factors

Figure 7-2: Two probabilistic programs with the same marginal distribution (shown in the final panel in Figure 7-1).

7.2.2 Sequential Monte Carlo

Suppose our target distribution is X with probability mass function p. Importance sampling generates samples from an approximating distribution Y (with probability mass function q) and re-weights the samples to account for the difference between true and approximating using $w(x) = p(x)/q(x)$. If we are interested in some expectation $\psi = \mathbb{E}_{x \sim X}[f(x)]$, we can estimate it from samples $y_1, \ldots, y_n$ using $\hat{\psi} = {\sum_{i=1}^{n} w(y_i) f(y_i)}/{\sum_{i=1}^{n} w(y_i)}$. To generate approximate samples from $p(x)$, we resample from the set of weighted samples in proportion to the importance weights.

If we iterate this procedure with a sequence of approximating distributions $q_1, \ldots, q_k$, we get *Sequential Importance Sampling.* If we resample at each stage, we get *Sequential Importance Resampling.* If we additionally apply MCMC "rejuvenation" steps at

each stage i with a transition kernel that leaves the distribution q_i invariant, we get *Sequential Monte Carlo*.

For Sequential Importance Sampling, the sum of the KL divergences between successive distributions controls the difficulty of sampling [18]. If we can sample from the right coarse-grained distributions, we can reduce this difficulty, as illustrated in Figure 7-1. With rejuvenation steps (SMC), the picture is more complex, but empirically, it is still the case that distributions that are closer together in KL generally make the sampling problem easier. In particular, we expect that good coarse-to-fine sequences lead to better coverage of regions with high posterior probability, and that they enable more efficient pruning of low-probability regions. A finite set of fine-grained particles may not cover the entire region, which can lead to a situation where all particles assign low probability to the next filtering step (particle decay). A particle that has not been refined yet corresponds to distributions on fine-grained states, thus each such particle can cover a bigger region [80]. Good coarse-to-fine sequences can allow us to prune entire parts of the state space, only considering refinements of abstract states that have sufficiently high posterior probability [42].

7.3 Algorithm

Given a probabilistic program, our algorithm builds a coarse-to-fine program with the same marginal distribution as the original program, but with additional latent structure corresponding to coarsened versions of the program.

We will assume that the user provides a `coarsenValue` function that describes how values map to more abstract values. Iterating this function leads to levels of coarsened values. Our goal then is to construct a version of the original program that operates over values coarsened N times. We will preserve the basic flow structure of

```
var liftedUniformERP = liftERP(uniformERP)
var liftedDistance = liftScorer(distance)

var noisyObserve = f(obs){
  var score = -3*liftedDistance(obs, 7)
  liftedFactor(score)
}

var model = f(){
  store.base = getStackAddress()
  var x = liftedUniformERP()
  var y = liftedUniformERP()
  var observation = flip(0.5) ? x : y
  noisyObserve(observation)
  return [x, y]
}

var coarseToFineModel = f(level){
  store.level = level
  var marginalValue = model()
  if (level === 0) {
    return marginalValue
  } else {
    return coarseToFineModel(level - 1)
  }
}
```

Figure 7-3: The coarse-to-fine model corresponding to the model in Figure 7-2a. This model has the same marginal distribution as 7-2a and 7-2b, but samples it using the hierarchical process shown in Figure 7-1. The functions `liftERP`, `liftScorer`, and `liftedFactor` are described in Sections 7.3.5-7.3.7.

the program, and thus we only need to specify how each primitive construct in the program is lifted to the space of coarsened values. The tricky part is to construct these lifted components such that the final marginal distribution is preserved. We use two ideas to accomplish this. First, we replace each unconditional elementary distribution at a given location with a distribution that depends on the coarser value of the same location, but such that marginalizing out this coarser value yields the original distribution. Second, we treat lifted factors as only approximations useful for guiding inference, which are then canceled by an extra factor inserted at the next-finer level. With this scheme, only the finest-level factors contribute to the final score. This scheme gives us flexibility over the lifting of primitive functions: values of lifted primitive functions flow to heuristic factor statements, not to the program's return statement, hence lifted functions only need to have similar behavior to their original; deviations will be corrected by the cancellation of factors.

In the next few subsections, we first introduce heuristic factors, then the inputs that the program transform requires, how the model syntax is transformed, and how each of the components of the lifted model works: constants, random variables, factors, and primitive functions.

7.3.1 Heuristic factors

A *heuristic factor* is a factor that is introduced for the purpose of guiding incremental inference algorithms such as particle filtering and best-first enumeration [24]. Its distinguishing characteristic is that an equivalent, canceling factor is inserted at a later position in the program in order to leave the program's distribution invariant. In other words, the pair of statements `factor(s)` and `factor(-s)` together has no effect on the meaning of a model; its only effect is in controlling how inference algorithms

explore the state space.

For example, Figure 7-2b shows a way to rewrite the program in Figure 7-2a in a way that initially assigns higher weight to program executions where x is close to the true observation 7. This is a heuristic, since—depending on the outcome of the coin flip—it may be y which is observed, in which case there is no pressure for x to be close to 7.

We will use heuristic factors to guide sampling on coarse levels towards high-probability regions of the state space without changing the program's distribution.

7.3.2 Prerequisites

The main inputs to the transform are a model, given as code for a probabilistic program, and a pair of functions `coarsenValue` and `refineValue`.

The main constraint on the model is that all ERPs need to be independent, i.e., do not take parameters that depend on other ERPs. If the support of each ERP is known, this can be achieved using a simple transform that replaces each dependent ERP with a maximum-entropy ERP, and adds a factor that corrects the score. That is, we transform

```
var x = sample(originalERP, params)
```

to

```
var x = sample(maxentERP)
factor(originalERP.score(x, params) -
       maxentERP.score(x))
```

This transform leaves the model's distribution unchanged and greatly simplifies the coarsening of ERPs, but reduces the statistical efficiency of the model. This statistical inefficiency can be addressed by merging `sample` and `factor` statements

into `sampleWithFactor` after the coarse-to-fine transform [24].

The model is annotated with the name of the main model function (which defines the marginal distribution of interest) and a list of names of ERPs, constants, and functions to be lifted (compound, primitive, score, and polymorphic; see below).

The main parameters that control the coarse distributions are the user-specified functions `coarsenValue` and `refineValue`. The function `coarsenValue` maps a value to a coarser value; the function `refineValue` maps a coarse value to a set of finer values. To generate values on abstraction level i, we iterate the `coarsenValue` function i times. We require that the two functions are inverses in the sense that $v \in \texttt{refineValue}(V) \Leftrightarrow \texttt{coarsenValue}(v) = V$ for all v and V.

This chapter does not address the task of finding good value coarsening functions. Instead, we ask: if such a function is given, how can we use it to coarsen entire programs so that we produce a sequence of coarse-to-fine models that is useful for SMC?

7.3.3 Model transform

The transform adds a wrapper `coarseToFineModel` that calls the model once for each coarsening level, from coarse to fine, each time setting the (dynamically scoped) variable `store.level` (in the following, **level**) to the current level. The transform also replaces all ERPs, primitive functions, and score functions with lifted versions that act differently depending on **level**. The coarse models only affect the fine-grained models through the side-effect of storing the values of their random choices in `store`, which is used by the finer-grained models to conditionally sample *their* random choices.

The syntactic transform itself proceeds as follow:

1. For each ERP, primitive and score function, insert the corresponding lifted definition before the model definition. For example:

   ```
   var liftedPlus = liftPrimitive(plus)
   ```

2. Rename all ERPs, primitive and score functions to their corresponding lifted names in model and compound functions. For example, replace `plus` with `liftedPlus`.

3. Wrap all constants. For example, replace c with `liftConstant(`c`)`.

4. As the first statement in the model, store the current **address**, which is needed to compute relative addresses of random choices and factors later on:

   ```
   store.base = getStackAddress()
   ```

5. Add a wrapper `coarseToFineModel` that calls the model once for each coarsening level (see Figure 7-3).

We will now describe the mechanisms behind lifted constants, random variables, factors, and primitive functions.

7.3.4 Lifting constants

To lift a constant, we simply repeatedly coarsen it to the current level:

Algorithm 7-1: Lifting constants

procedure LIFTEDCONSTANT(c)
  for i=0; i < **level**; i++ **do**
    c = coarsenValue(c)
  end for
  return c
end procedure

Algorithm 7-2: Lifting ERPs

```
procedure SAMPLELIFTEDERP(e_0, l)
  v_1 = store[erpName(address, l + 1)]
  if v_1 is undefined then
    if l is 0 then
      return e_0.sample()
    else
      return coarsenValue(sampleLiftedERP(e_0, l − 1))
    end if
  else
    v⃗ = refineValue(v_1)
    p⃗ = v⃗.map(λ(v){return getERPScore(e_0, v, l)})
    return sampleDiscrete(v⃗, p⃗)
  end if
end procedure

procedure LIFTERP(e_0)
  e_1 = makeERP(λ(){SAMPLELIFTEDERP(e_0, level)})
  return λ(){
    v = sample(e_1)
    store[erpName(address, level)] = v
    return v
  }
end procedure
```

7.3.5 Lifting random variables

Let D_0 be the domain of an ERP with distribution $p_0(x)$, and let $D_n = \mathrm{cv}^n(D_0)$ be the set of values arrived at by repeatedly applying the `coarsenValue` function (written cv for short). We would like to decompose the original distribution $p_0(x)$ into a sequence of conditional distributions $q(x_n|x_{n+1})$ for random variables $x_n \in D_n$. If we take:

$$q(x_n|x_{n+1}) \propto \sum_{x_0} p(x_0)\mathbb{1}_{\mathrm{cv}^n(x_0)=x_n \wedge \mathrm{cv}(x_n)=x_{n+1}}$$

and for the coarsest level, N,

$$q(x_N) = \sum_{x_0} p(x_0) \mathbb{1}_{\mathrm{cv}^N(x_0)=x_N}$$

then it is clear that we preserve the marginal distribution on x_0. That is:

$$p(x_0) = \sum_{x_1,\ldots,x_N} q(x_0|x_1) \cdots q(x_{N-1}|x_N) q(x_N).$$

Algorithm 7-2 shows how we implement sampling from such a decomposed ERP at a given level. Note that to lookup the existing value at the next-coarser level we identify random variables on different levels based on relative stack addresses (via `erpName`); this is critical for models such as grammars with an unbounded number of random choices. The implementation for computing the score of lifted ERPs is analogous (although this score is not needed for pure particle filtering, so we omit the details). Both are parameterized by a function `getERPScore` that estimates the total probability of the equivalence class of values that map to a given coarse value. These ERP scores for coarse values can be estimated by sampling refinements, via user-specified scoring functions (as in Section 7.4.1), or using exact computation (as in Section 7.4.2).

7.3.6 Lifting factors

We treat the lifted counterpart to factors as heuristic factors: the score on the next-higher level is subtracted out on the current level. By canceling out these factors when inference proceeds to finer-grained levels, we ensure that the overall distribution of the model remains unchanged—ultimately, only the base-level factors count. This incremental scoring process formalizes the intuition of increasing attention to detail

as we move down the abstraction ladder. Like random variables, we identify factors based on relative stack addresses.

Algorithm 7-3: Lifting factors

> **procedure** LIFTEDFACTOR(s)
> s_1 = store[factorName(**address**, **level** + 1)] $\vee$ 0
> factor($s - s_1$)
> store[factorName(**address**, **level**)] = s
> **end procedure**

7.3.7 Lifting primitive functions

When a primitive function f is applied to a base-level value, it is deterministic. Now we are interested in lifting primitive functions to operate on coarse values. However, each coarse value corresponds to a set of base-level values. For different elements in this set, f may return different values. This suggests that lifted versions of f may be stochastic. We wish to preserve the marginal distribution of the entire program, but since we treat coarse factors as canceling heuristics, we have some latitude in how to lift the primitive functions.

Algorithm 7-4 shows one approach to the computation of such coarsened primitives. This algorithm is parameterized by a function `marginalize`, which may be implemented using exact computation, sampling, etc., and which may cache its computations.

Scoring functions—i.e., functions that directly compute a score to be consumed by `factor`—are a special case of primitive functions. We know that they return a number, so instead of sampling from the return distribution, we can simply compute the expected value. We find that this leads to more stable, and hence useful, heuristic factors.

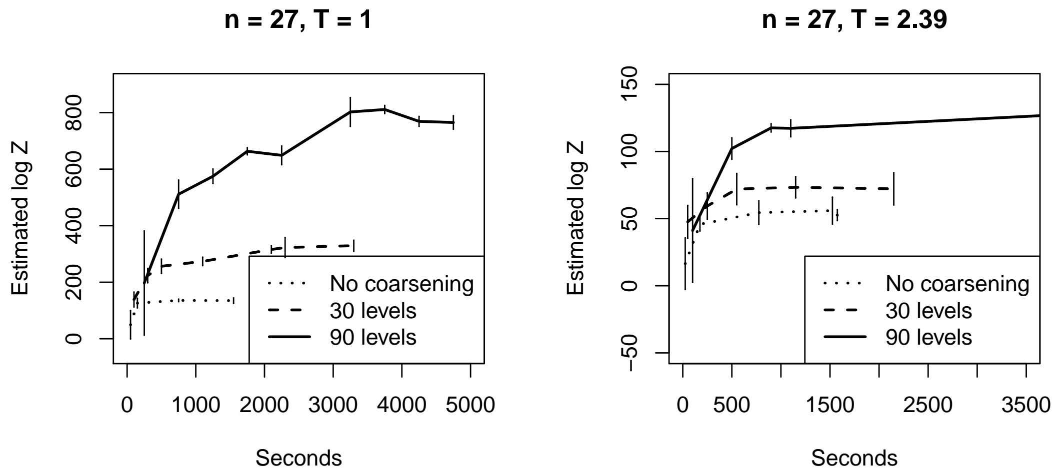
Algorithm 7-4: Lifting primitive functions

```
procedure liftPrimitive(f)
  return λ(x⃗){
    e = marginalize(λ(){
      x⃗_0 = x⃗.map((uniformDraw ∘ refineValue)^level)
      v = f(x⃗_0)
      return coarsenValue^level(v)
    })
    return sample(e);
  }
end procedure
```

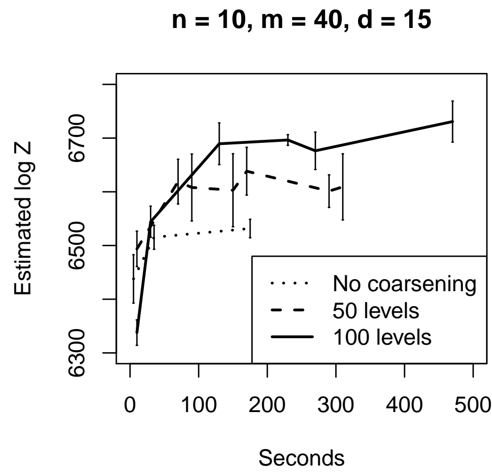
Algebraic data type constructors are another special case. In many circumstances (including Section 7.4.2), they can be treated as transparent with respect to coarsening. For example, it is frequently useful to define `coarsenValue(`[x_1, x_2, ...]`)` as equivalent to [`coarsenValue(`x_1`)`, `coarsenValue(`x_2`)`, ...]. More generally, if a function can apply to coarsened objects directly, it can be marked as *polymorphic*. In that case, no lifting is necessary. For example, if we have a function that computes the mean of a list of numbers, and if our coarsening maps lists of numbers to shorter lists of numbers without changing their type, then we have the option to mark this function as polymorphic.

7.4 Empirical evaluation

To evaluate the proposed technique, we apply it to three classes of models in the domain of tracking partially observable objects over time given visual information: first, an Ising model and its depth-from-disparity variation; second, a factorial hidden Markov model for multiple object tracking; and third, a simple model of visual scene understanding.



(a) Ising at low temperature ($T = 1$) (b) Ising at the critical state ($T = 2.39$)



(c) Depth-from-disparity

Figure 7-4: Quantitative inference results for Markov Random Field models

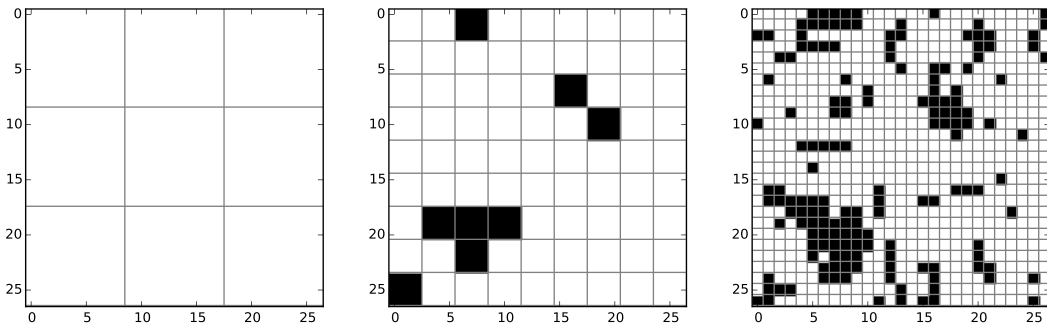


Figure 7-5: Coarsening the Ising model at the critical state ($T = 2.39$)

7.4.1 Markov Random Fields

A number of applications in physics, biology, and computer vision can be modeled as Markov Random Fields (MRFs). These problems have in common that they specify a global energy function which, by virtue of the Markov property, depends only on the local neighborhoods of elements. Once this energy function is specified, it can be difficult to minimize; specialized optimization algorithms have been developed for particular domains [87], but there is no generally applicable solution.

The local neighborhood structure of the energy function, however, suggests that a coarse-to-fine transformation may be useful: if neighborhoods are coarsened into single representative values, then the energy can be minimized in this smaller space, using heuristic factors to guide search in the original space. We do not claim that our model transformation constitutes a solution in itself, but can be used in tandem with other algorithms to effectively reduce the search space. Here, we demonstrate the coarse-to-fine transformation on two simple MRFs: the Ising model and the stereo matching task.

Ising model

Coarse-to-fine transformations have a long history of applications in physics. When studying systems which interact across multiple orders of magnitude, such as fluids, ferromagnets, and metal alloys, it is intractable to work at the most fine-grained level. Since exact solutions do not exist, physicists developed a method called the *renormalization group* [97, 98], which effectively maps the fine-grained representation of a system onto a coarser but identically parameterized representation with similar properties.

One of the simplest testbeds for renormalization group methods is the 2-dimensional Ising model. The state space is an $n \times n$ lattice of cells, each of which can take one of two spin values, $\sigma_i \in \{-1, +1\}$. The *energy* of a particular configuration of spins σ is given by the Hamiltonian:

$$\mathcal{H}(\sigma) = J \sum_{\langle ij \rangle} \sigma_i \sigma_j$$

where J is the *interaction constant* and $\langle ij \rangle$ indicates summing over all possible pairs of neighbors. Note that the number of possible configurations grows exponentially in n, rendering an exhaustive search for low-energy states impossible.

The interaction constant can be written $J = 1/T$ where T is the *temperature* of the system. The configuration distribution takes different forms at different temperatures: we will conduct experiments at $T = 1$, a low-temperature condition where spins prefer to globally align, and $T = 2.39$, the *critical temperature*, where long-range correlations dominate. Above the critical temperature (e.g. for $T > 3$), cells become uncoupled and the energy distribution over configurations converges to uniform, so we focus on lower temperatures.

We implemented the Ising energy-minimization problem as a simple probabilistic program, which first samples a set of spins and then factors based on the energy of that configuration. To apply our coarse-to-fine transformation, we used the spin-block majority-rule for our `coarsenValue` and `refineValue` functions. To coarsen, this rule replaced each 3×3 sub-lattice with its modal value (see Figure 7-5). To refine a single cell in the coarse matrix, we considered the space of all 256 possible 3×3 matrices that could coarsen to that value. Note that this sequential refinement – making many small choices instead of one big one – differs in interesting ways from the typical renormalization group approach, which simultaneously replaces all sub-lattices and reweights the interaction constant J accordingly. To facilitate sequential refinement, we implemented a polymorphic energy function, which can directly evaluate partially coarsened matrices without needing to refine all the way down to the base.

We ran two experiments on 27×27 lattices to demonstrate the performance of our coarsened program at different levels of coarsening. In our first experiment, we set the temperature to $T = 1$ and averaged 10 runs for both flat filtering and coarse-to-fine filtering (with 30 and 90 levels). The 90 level coarsening condition fully reduces the 27×27 lattice to a 3×3 lattice, and the 30 level condition yields a partially coarsened matrix. In our second experiment, we set the temperature to $T = 2.39$ and run the same set of conditions. Figures 7-4a and 7-4b show the average importance weight for different levels of coarsening. We see that even intermediate levels of coarsening perform better than flat filtering, and that a full coarsening performs dramatically better than the other conditions . The best solution from the second experiment is shown in the right-most panel of Figure 7-5, along with the two coarser configurations from which it was refined. This displays the characteristic local-neighborhood structure of the Ising model at critical temperatures.

Stereo matching

Another common application of MRFs is the stereo matching task [8, 73]. The goal is to estimate the disparity between two images, $\mathcal{I}$ and $\mathcal{I}'$, captured from slightly shifted viewpoints. This disparity map can be used to recover a rough measure of depth. As in the Ising case, we implement this task in a probabilistic programming language by sampling a lattice of disparity values, and factoring on its energy.

The energy function for a particular set of disparity values has two parts: (1) a *smoothing* term penalizing distance between the values of neighbors and (2) a *data cost* term penalizing each particular disparity value for discrepancies with the true data (as measured by comparing the difference in pixel intensities at the given discrepancy):

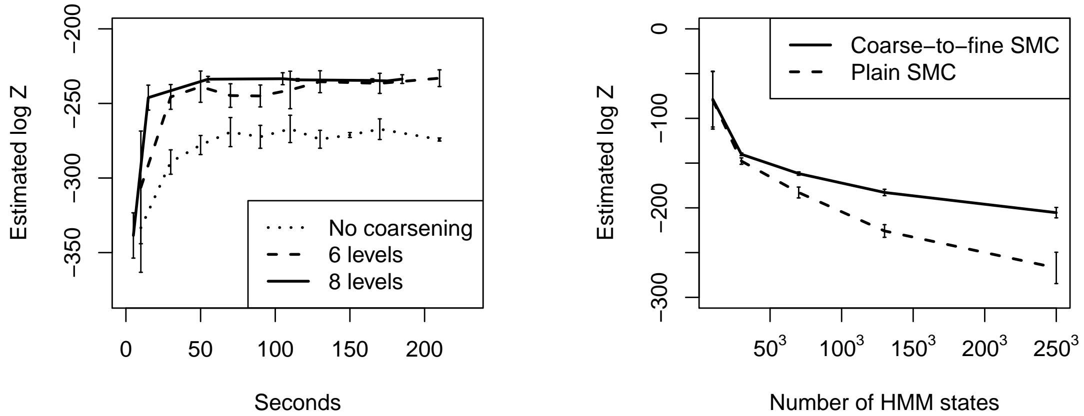
$$\mathcal{H}(d) = \sum_{\{p,q\}\in\mathcal{N}} V_{p,q}(d_p, d_q) + \sum_{p} C(p, d_p)$$

We denote the intensity of pixel p in image $\mathcal{I}$ by $\mathcal{I}_p$. Since corresponding pixels should have similar intensities, we set our data cost term $C(p, d_p)$ as suggested by [8], taking the absolute difference between $\mathcal{I}_p$ and $\mathcal{I}'_{p+d_p}$. To reduce sensitivity to variability in image sampling, we interpolate between neighboring intensities in the neighborhood $x \in (d - 0.5, d + 0.5)$ and take the minimum. For our smoothing function, we use the truncated squared error:

$$V_{p,q}(d_p, d_q) = \min((d_p - d_q)^2, V_{\max})$$

with $V_{\max} = 5$.

We implemented energy minimization for the stereo matching model analogous to the Ising model. Figure 7-4c shows that in a comparison between coarse-to-fine



(a) SMC using a coarse-to-fine model finds more probable samples earlier on than SMC without coarsening.

(b) For small numbers of states, coarse-to-fine is indistinguishable from plain particle filtering. As the number of states grows, coarse-to-fine is able to provide better solutions in the same amount of time.

Figure 7-6: Inference results for a factorial HMM

SMC and importance sampling, SMC finds lower-energy states more efficiently for a 10x40 cropped pair of images from the Middlebury dataset [73], although the absolute quality of the states found is difficult to evaluate from these numbers.

7.4.2 Factorial HMM

In our second example, we test the hypothesis that abstractions are useful as a means to avoid particle collapse in large state spaces. For this reason, we chose the factorial HMM, a model with a large effective state size even within a single particle filter step. The Factorial HMM is a HMM where the state factors into multiple variables [20]. If there are M possible values for each latent state, and k state variables per time

step, then the effective state size is M^k. If only few of these have high probability, then even for moderate M and k it is possible that there are not sufficiently many fine-grained particles to cover all regions of high posterior probability.

In our first experiment, we use a factorial HMM with 3 variables per step, 256 possible state values per variable, and 6 observed time steps. We run both coarse-to-fine filtering (with 6 and 8 abstraction levels) and flat filtering, and average 10 runs.

We coarsen the factorial HMM by merging some state and observation symbols. To test the hypothesis that coarse-to-fine inference will work best when abstractions match the dynamics of the model, we generate transition and observation matrices with approximately hierarchical structure as follows. Enumerate state 1 to N. For states i and j, we let the transition probability be approximately proportional to $2^{-|i-j|}$. Similarly, for state i, the probability of generating observation k is proportional to $2^{-|i-k|}$.

Figure 7-6a shows the average importance weight, which is an estimate for the normalization constant. Since such estimates are essentially lower bounds [31], we can conclude that plain particle filtering consistently underestimates the true normalization constant.

In our second experiment (Figure 7-6b), we compare the behavior of flat and coarse-to-fine filtering as the number of HMM states increases from 2^3 to 256^3. As before, we keep the runtime constant. For small numbers of states, plain and coarse-to-fine SMC give very similar estimates. As the number of states grows, the difference between coarse-to-fine and plain filtering grows as well, indicating that coarse-to-fine is most useful in large state spaces.

(a) Ground truth (b) Inferred layout

Figure 7-7: Visual scene understanding results

7.4.3 Visual scene understanding

Vision is a prime candidate for coarse-to-fine approaches: data is high-dimensional, fast recognition performance is frequently desired, and the hierarchical organization of our visual cortex—starting with low-level concepts such as edges, progressing towards areas that process objects and motions—suggests that the human brain may itself employ a similar approach.

In this final qualitative experiment, we test the coarse-to-fine approach on the problem of recovering the objects underlying a scene given a pixel-based image. Specifically, we look at the kinds of entities one might see in satellite photography: houses, lawns, and swimming pools, seen from above. Given a generative model that selects such objects and their types, coordinates and angles, and renders an image, the task is to invert this model and determine the objects and their properties given an image.

Figure 7-7 shows an example. The first panel shows the true image, the second shows the result of recovering object data by applying particle filtering to a three-level coarse-to-fine model, and re-rendering the object properties onto an image.

7.5 Discussion

When does coarse-to-fine inference help? It is generally difficult to compute or estimate the posterior probability of a set of (program) states. However, precisely this is required for coarse-to-fine inference to work: when we evaluate a program on a coarse level, we need to estimate for each coarse value how likely its refinements are under the posterior. This suggests that settings where coarse-to-fine inference works have special characteristics that make such estimation feasible. We now name a few.

First, the given program may satisfy independence assumptions that make estimating posterior probabilities feasible. For example, for the program shown in Figures 7-1 and 7-2a, the score function only depends on one of x and y at a time; hence, we can independently compute the estimated score for the refinements of x and y, and use this information in computing the estimated scores for abstract values of both. *Second*, we may be in a setting where the type of a coarse value matches the type of its refinements. In that situation, "polymorphic" score- and primitive functions may be a cheap way to estimate the posterior probability of a coarse state. For the Ising model, the energy function satisfies this criterion up to parameterization. *Third*, coarse-to-fine may be particularly useful in the amortized setting [85]. Learning the conditional distributions associated with lifted primitive functions is one instantiation of "learning to do inference". This is particularly feasible for smooth state spaces, where one can effectively estimate entire distributions from a few samples.

Another answer to the question of when coarse-to-fine helps is to point out that this depends on what inference algorithm is used. For inference by enumeration, exact coarsening (i.e., coarsening within values that have the *same* posterior probability) is useful for increasing computational efficiency. By contrast, for sequential Monte

Carlo methods, it is frequently more desirable to coarsen together states with *different* posterior probability, as it smoothes the state space and thus increases statistical efficiency.

The work in this chapter is related to and inspired by a broad background of work on coarse-to-fine and lifted inference, including (but not limited to) work by Charniak et al. [10] on coarse-to-fine inference in PCFGs, work on coarse-to-fine inference for first-order probabilistic models by Kiddon and Domingos [42], and attempts to "fatten" particles for filtering with broader coverage [48, 80]. Outside of machine learning, we take inspiration from *Approximate Bayesian Computation* (conditioning on summary statistics) and *renormalization group* approaches to inference in Ising models [52].

This approach opens up many exciting research directions, including: understanding the relation to abstract interpretation, and Galois Connections specifically [e.g., 12, 57]; automatically deriving coarsenings for hierarchical Bayesian models; learning good coarsenings, and efficient learning of approximations for coarsened primitive and score functions; and coarsening (merging) multiple variables across program blocks, potentially via flow analysis.

We expect that the most interesting applications of coarse-to-fine approaches to efficient inference are yet to come.

Chapter 8

The Road Ahead

This thesis set out to show that probabilistic programs are a productive metaphor for understanding how the mind works. What have we learned, and what is there yet to be learned?

After a brief introduction of probabilistic programs in Chapter 2, Chapters 3 and 4 illustrated the representational power of probabilistic programs. I proposed *generative* probabilistic programs as a representation language for concepts, and probabilistic inference as a mechanism for learning and reasoning with such concepts. I provided examples of richly structured concepts, defined in terms of systems of relations, subparts, and recursive embeddings, that are naturally expressed as programs. I gathered initial experimental evidence that human generalization patterns for observations generated by such concepts can be explained using probabilistic programs, but are not naturally accounted for using previous methods. I then proceeded to models of reasoning about reasoning, a domain where the expressive power of probabilistic programs was necessary to formalize our intuitive domain understanding. This is due to the fact that, unlike previous formalisms, probabilistic programs allow

conditioning to be *represented in* a model, not just *applied to* a model. I illustrated this insight using programs that concisely express reasoning about agents in game theory, artificial intelligence, and linguistics.

In Chapters 5-7, we looked at three inference algorithms with the dual intent of showing how to efficiently compute the marginal distributions defined by probabilistic programs and providing building blocks for eventual process-level accounts of human cognition. I developed a Dynamic Programming technique that can help make inference tractable in models where computation is reused; as a particularly important case, this includes the models of reasoning about reasoning that we saw in Chapter 4. I then started to address the puzzle of how humans can be so quick in recognizing words, objects, and scenes, while coherent probabilistic inference tends to be slow, frequently taking many thousands of sequential operations. I introduced the setting of *amortized inference* and showed that learning "recognition models" from past inference results can speed up future inferences. Finally, I described a generic approach to coarse-to-fine inference in probabilistic programs and showed evidence that it can speed up inference in models with large hierarchical state spaces.

Over the course of this thesis, we have gathered a few pieces of the puzzle of cognition, and in outline we can see how they might form part of a coherent solution. This coherence is by construction—the pieces are expressed within the common framework of probabilistic programs, after all. Finding the connections that make these pieces snap together is one of the major next challenges for understanding cognition in this framework. In the next few paragraphs, I will point out some of the connections we can already foresee, starting with the domain of concept learning and branching out to other topics covered in this thesis. In connecting the pieces, we have to be prepared that the number of required *additional* pieces may outnumber what we have so far, maybe vastly. Indeed, we do not know the size of the emerging

picture; there may be entire regions that we have never even considered, because we have not yet built up the conceptual vocabulary required to think about them. While it is naturally difficult to know where such blind spots are, I will highlight some parts that currently look blurred under the lens of probabilistic programming; some parts that the streetlamp we are searching under may only partially illuminate. But first, let us think about phenomena that are well within the explanatory reach of this framework.

The model of concept learning presented in Chapter 3 is a caricature. I have argued that it is a *good* caricature, one that captures key parts of human concept acquisition, but it is a caricature nonetheless, and for several reasons. *First*, much of human learning is social: children learn from interactions with their parents, and students learn from interactions with their teachers. We are not passive recipients of randomly sampled data; instead, we are active participants in the learning process, choosing how to interact with our teachers and the world. *Second*, we learn from language. For example, Lupyan et al. [51] show that verbal labels help human subjects learn to classify objects into categories, even when the labels are redundant. Cabrera and Billman [9] investigated how "nonsense" sentences such as "the mugglet troces the diggy" and "the mugglet and the diggy troce" affect the learning of event categories. *Third*, Schulz [75] argues that, in contrast to current approaches to inference in probabilistic programs that are commonly based on random walks and other "uninformed" sampling techniques, children's learning seems to be more constructive and directed. *Fourth*, as useful as the separation between representation and inference is for clarifying our understanding of human concept learning, it is almost certain that conceptual change and learning to reason with new concepts go hand in hand, and that more comprehensive accounts of concept learning will have something to say about this interaction.

Fortunately, we can already foresee how we might extend the program induction model in Chapter 3 to account for some of these phenomena. Models of reasoning about reasoning as described in Chapter 4 are a natural component for future models of concept learning, including models of learning in pedagogical situations [77] and models that combine language and concept learning [e.g., 17]. Indeed, recent work by Hawkins et al. [34] has developed probabilistic programs that combine active learning and natural pedagogy in a model of question-answering; a very similar model could be used to study concept learning.

For inference in such models, similarly straightforward extensions are possible. When we acquire a new concept, one of the challenges we face is in learning how to reason using this concept. We have proposed the setting of amortized inference, and specifically the learning of inverse recognition models, as a candidate for what "learning to reason" could mean. The integration of concept learning and "inverse learning"—maybe in a wake-sleep style algorithm [36]—is an exciting direction for future work. Similarly, coarse-to-fine approaches seem to be useful not only for inference given a model, but also for learning a model in the first place. That is, it is plausible that children first learn a coarse model of the world before they fill in the details. Understanding how this kind of learning works, and how it interacts with coarse-to-fine inference, is a natural direction, as is the follow-up idea of learning inverses for coarse-to-fine models.

There are plenty of other future directions for work on inference, including: (1) Can we use counterfactuals and responsibility attribution mechanisms to determine which parts of a model are to blame if a model fails to explain the data well? (2) How can we implement probabilistic programs using distributed computation? As a particularly interesting instance, can we compile probabilistic programs so that they run on a substrate provided by a (artificial) neural net? Does this constrain

the kinds of probabilistic programs we can write? (3) Can we find classes of programs where inference is always easy, or likely to be easy? Can we define priors that systematically favor such classes? (4) What is a systematic general approach to caching computation in probabilistic programs? In Chapters 5 and 6, I describe two approaches. Can we unify them under a single mechanism that makes principled decisions about what computations to reuse? None of these questions are trivial, some could almost certainly be a thesis in their own right, but most importantly, all seem productively approachable at this point.

Are there any phenomena that are *not* a good fit for probabilistic programming? The answer is almost certainly *yes*, but the generality of the framework makes it difficult to be more specific. For any phenomenon, it is likely that we can come up with a computational treatment, and if one does not readily come to mind, we cannot easily distinguish inadequacy of the framework from failures of imagination. An answer to this question will probably only emerge as a hard-won empirical insight that some classes of phenomena resist formalization in this framework more than others. That said, it is still worthwhile to ask: In what ways are human concepts *unlike* programs? In what ways is thinking *unlike* Bayesian inference? I do not have good answers to these questions, but one of the topics I would investigate as a candidate for an idea that might be challenging to tackle using probabilistic programs is the notion of *logical uncertainty* [e.g., 33]; that is, uncertainty that can be resolved by more reasoning, without additional observations. On the side of cognitive science, this comes up when we ask how our framework can accommodate the kind of thinking that mathematicians regularly do. In computer science, this comes up when we think about meta-inference—that is, the idea of treating inference mechanisms as the objects to be reasoned about. A good mechanism for choosing how to do inference in any given model may itself employ inference. What would

such meta-level probabilistic programs look like?

Other questions that could test the explanatory power of our framework include: (1) How do the relatively abstract representations discussed here relate to the more primitive building blocks of the human mind (such as emotions and reinforcement learning), which may in part be shared with our animal ancestors? (2) We have discussed how to model the acquisition of beliefs and concepts. Can we similarly model the acquisition of values? How does joint learning of values and concepts play out when values themselves are expressed in terms of concepts that are in flux? (3) Humans may have different overlapping systems of concepts for understanding a single domain, and the conclusions drawn from one such system may only partially propagate to the others. How can we model this multiplicity of views? By focussing on questions like these, which are arguably close to the boundary of our framework, we may be able to better understand where exactly this boundary lies.

For now, it is difficult to predict how far the framework of probabilistic programs will carry us. Over the last decades, computation and probability have turned out to be solid building blocks for the study of cognition. It is unlikely that either will be displaced soon, and their combination in probabilistic programs is both natural and productive, as I hope this thesis has demonstrated. It is not outlandish to think that, while the road ahead is long, this framework will carry us all the way, without fundamental revision. Or maybe there will come a point when most of what there is to learn within this framework has been learned; when both its potential and its limitations are well-understood, and a pattern in its limitations emerges that points towards a fundamentally new way of studying cognition. This, too, is not outlandish to think. The truth is likely in between, and it will be exhilarating to see both the ways in which probabilistic programming will form a lasting building block for the science of cognition, and the ways in which new concepts will be required.

Bibliography

[1] Harold Abelson, Gerald J Sussman, and Julie Sussman. *Structure and interpretation of computer programs, (second edition)*, volume 33. MIT Press, Cambridge, MA, USA, 2nd edition, 1997. ISBN 0-262-01153-0.

[2] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*, volume 1009. Addison Wesley, 2007.

[3] John R. Anderson. The adaptive character of thought. *Hillsdale NJ Earlbaum*, 104:276, January 1990. ISSN 0002-9556.

[4] Chris Baker, Rebecca Saxe, and Joshua B. Tenenbaum. Bayesian models of human action understanding. *Advances in Neural Information Processing Systems*, 18:99–106, 2005.

[5] Chris L Baker, Rebecca R Saxe, and Joshua B Tenenbaum. Bayesian Theory of Mind. *Proceedings of the Thirty-Second Annual Conference of the Cognitive Science Society*, 1:1–10, 2011.

[6] Paul Bello. Cognitive Foundations for a Computational Theory of Mindreading. *Advances in Cognitive Systems*, 2012.

[7] Paul Bello and Nick Cassimatis. Developmental accounts of theory-of-mind acquisition: Achieving clarity via computational cognitive modeling. *28th Annual Conference of the Cognitive Science Society, Vancouver, Canada*, 2006.

[8] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(11):1222–1239, 2001.

[9] Ángel Cabrera and Dorit Billman. Language-driven concept learning: Deciphering Jabberwocky. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 22(2):539, 1996.

[10] Eugene Charniak, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R. Shrivaths, Jeremy Moore, Michael Pozar, and Theresa Vu. Multilevel Coarse-to-fine PCFG Parsing. *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 168–175, 2006.

[11] Jian Cheng and Marek J. Druzdzel. AIS-BN: An Adaptive Importance Sampling Algorithm for Evidential Reasoning in Large Bayesian Networks. *Journal of Artificial Intelligence Research*, 13:155–188, 2000. ISSN 10769757.

[12] Patrick Cousot and Michael Monerau. Probabilistic abstract interpretation. In *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 169–193. Springer, 2012.

[13] Jason Eisner, Eric Goldlust, and Noah A Smith. Compiling Comp Ling: Practical Weighted Dynamic Programming and the Dyna Language. In *Proceedings of Human Language Technologies and the Conference on Empirical Methods in Natural Language Processing*, 2005.

[14] Jacob Feldman. The Structure of Perceptual Categories. *Journal of mathematical psychology*, 41:145–70, January 1997. ISSN 0022-2496.

[15] Jerry Alan Fodor. *The Language of Thought*, volume 5. Harvard University Press, 1975. ISBN 9780674510302.

[16] Michael C. Frank and Noah D. Goodman. Predicting Pragmatic Reasoning in Language Games. *Science*, 336(6084):998–998, May 2012. ISSN 0036-8075.

[17] Michael C. Frank, Noah D. Goodman, and Joshua B. Tenenbaum. Using speakers' referential intentions to model early cross-situational word learning: Research article. *Psychological Science*, 20(5):578–585, 2009. ISSN 09567976.

[18] Cameron E Freer, Vikash K Mansinghka, and Daniel M Roy. When are probabilistic programs probably computationally tractable? *NIPS 2010 Workshop on Monte Carlo Methods in Modern Applications*, 2010.

[19] George Gamow and Marvin Stern. Puzzle-math, 1958.

[20] Zoubin Ghahramani and Michael I Jordan. Factorial Hidden Markov Models. *Machine Learning*, 29(2-3):245–273, November 1997.

[21] Valerie Goffaux, Judith Peters, Julie Haubrechts, Christine Schiltz, Bernadette Jansma, and Rainer Goebel. From coarse to fine? Spatial and temporal dynamics of cortical face processing. *Cerebral Cortex*, page bhq112, 2010.

[22] Joshua Goodman. Semiring Parsing. *Computational Linguistics*, 25(4), 1999.

[23] Noah D Goodman and Andreas Stuhlmüller. Knowledge and implicature: Modeling language understanding as social cognition. *Topics in Cognitive Science*, 2013.

[24] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014.

[25] Noah D Goodman, Chris L Baker, Elizabeth Baraff Bonawitz, Vikash K Mansinghka, Alison Gopnik, Henry Wellman, Laura Schulz, and Joshua B Tenenbaum. Intuitive Theories of Mind : A Rational Approach to False Belief. *Cognitive Science Conference Proceedings*, pages 1382–1387, 2006.

[26] Noah D Goodman, Vikash Mansinghka, Daniel M Roy, K A Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence*, pages 220–229, 2008.

[27] Noah D Goodman, Joshua B Tenenbaum, Jacob Feldman, and Thomas L Griffiths. A rational analysis of rule-based concept learning. *Cognitive Science*, 32(1):108–154, 2008.

[28] Noah D Goodman, Joshua B Tenenbaum, Timothy J O'Donnell, and the Church Working Group. Probabilistic Models of Cognition, 2011.

[29] Alison Gopnik and Andrew N Meltzoff. *Words, thoughts, and theories.* MIT Press, 1998.

[30] Thomas L. Griffiths, Kevin R. Canini, Adam N. Sanborn, and Daniel J. Navarro. Unifying rational models of categorization via the hierarchical Dirichlet process. *Proceedings of the 29th Annual Conference of the Cognitive Science Society*, January 2009.

[31] Roger Grosse. Unbiased estimators of partition functions are basically lower bounds. `http://hips.seas.harvard.edu/`, January 2013.

[32] Heikki Haario, Marko Laine, Antonietta Mira, and Eero Saksman. DRAM: Efficient adaptive MCMC. *Statistics and Computing*, 16(4):339–354, 2006. ISSN 09603174.

[33] Joseph Y Halpern. *Reasoning about Uncertainty*, volume 21. MIT Press Cambridge, 2003.

[34] Robert X.D. Hawkins, Andreas Stuhlmüller, Judith Degen, and Noah D. Goodman. Why do you ask? Good questions provoke informative answers. In prep.

[35] Luis D Hernandez, Serafin Moral, and Antonio Salmeron. A Monte Carlo algorithm for probabilistic propagation in belief networks based on importance sampling and stratified simulation techniques. *International Journal of Approximate Reasoning*, 18(1):53–91, 1998.

[36] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. The "wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214): 1158–1161, 1995. ISSN 0036-8075.

[37] Berthold K P Horn. Understanding image intensities. *Artificial intelligence*, 8 (2):201–231, 1977.

[38] Laurence R Horn. Implicature. In *The Handbook of Pragmatics*. Blackwell Publishing Ltd, Oxford, UK, January 2006.

[39] Irvin Hwang, Andreas Stuhlmüller, and Noah D Goodman. Inducing Probabilistic Programs by Bayesian Program Merging. 2011.

[40] Edwin T. Jaynes. *Probability theory: the logic of science*, volume 27. Cambridge university press, 2005. ISBN 0521592712.

[41] Charles Kemp, Aaron Bernstein, and Joshua B. Tenenbaum. A generative theory of similarity. *Proceedings of the 27th Annual Conference of the Cognitive Science Society*, pages 1132–1137, January 2005.

[42] Chloé Kiddon and Pedro Domingos. Coarse-to-Fine Inference and Learning for First-Order Probabilistic Models. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, pages 1049–1056, August 2011.

[43] Dan Klein and Christopher D Manning. An O(n^3) Agenda-Based Chart Parser for Arbitrary Probabilistic Context-Free Grammars. Technical report, Stanford University, 1998.

[44] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[45] Daphne Koller and Brian Milch. Multi-agent influence diagrams for representing and solving games. *IJCAI International Joint Conference on Artificial Intelligence*, 45(1):1027–1034, 2001. ISSN 10450823.

[46] Daphne Koller, David McAllester, and Avi Pfeffer. Effective Bayesian Inference for Stochastic Programs. In *Proceedings of the National Conference on Artificial Intelligence*, pages 740–747, 1997.

[47] David M Kreps. A Course in Microeconomic Theory. *A Course in Microeconomic Theory*, pages xviii + 850, October 1990. ISSN 1935-990X.

[48] Tejas D Kulkarni, Ardavan Saeedi, and Samuel Gershman. Variational Particle Approximations. *arXiv preprint arXiv:1402.5715*, pages 1–11, 2014.

[49] Ugo Dal Lago and Margherita Zorzi. Probabilistic Operational Semantics for the Lambda Calculus. *RAIRO - Theoretical Informatics and Applications*, cs.LO(46):35, April 2011. ISSN 0988-3754.

[50] Hector J. Levesque. Thinking as Computation: A first course, 2012. ISBN 9780262016995.

[51] Gary Lupyan, David H Rakison, and James L McClelland. Language is not Just for Talking: Redundant Labels Facilitate Learning of Novel Categories. *Psychological Science*, 18(12):1077–1083, 2007.

[52] Edward Lyman and Daniel M. Zuckerman. Resolution exchange simulation with incremental coarsening. *Journal of Chemical Theory and Computation*, 2:656–666, 2006. ISSN 15499618.

[53] Arthur B Markman. *Knowledge representation.* Psychology Press, January 1999.

[54] David Marr. Vision: A computational investigation into the human representation and processing of visual information. *Henry Holt and Co., Inc. New York, NY, USA*, January 1982.

[55] Robert Mateescu, Kalev Kask, Vibhav Gogate, and Rina Dechter. Join-graph propagation algorithms. *Journal of Artificial Intelligence Research*, 37(1):279–328, 2010.

[56] Michael D Menz and Ralph D Freeman. Stereoscopic depth processing in the visual cortex: a coarse-to-fine mechanism. *Nature neuroscience*, 6(1):59–65, 2003.

[57] David Monniaux. Abstract interpretation of probabilistic semantics. In *Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 322–339. Springer, 2000.

[58] Quaid Morris. *Recognition Networks for Approximate Inference in BN20 Networks*. Morgan Kaufmann Publishers Inc., August 2001. ISBN 1-55860-800-1.

[59] Ian Murray and Zoubin Ghahramani. Bayesian learning in undirected graphical models: approximate MCMC algorithms. *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 392–399, 2004.

[60] Ian Murray, Zoubin Ghahramani, and David MacKay. MCMC for doubly-intractable distributions. *Proceedings of the 22nd Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 359–366, 2006.

[61] Shaun Nichols and Stephen P Stich. *Mindreading: An Integrated Account of Pretence, Self-Awareness, and Understanding Other Minds*. Oxford University Press, USA, October 2003.

[62] Robert M Nosofsky. Attention, similarity, and the identification-categorization relationship. *Journal of Experimental Psychology: General*, 115(1):39–57, 1986.

[63] Luis E Ortiz and Leslie Pack Kaelbling. Adaptive Importance Sampling for Estimation in Structured Domains. In *Proc. of the 16th Ann. Conf. on Uncertainty in A.I. (UAI-00)*, pages 446–454. Morgan Kaufmann Publishers, 2000.

[64] Slav Petrov, Aria Haghighi, and Dan Klein. Coarse-to-fine syntactic machine translation using language projections. *Empirical Methods in Natural Language*, pages 108–116, 2008.

[65] Avi Pfeffer. IBAL: A Probabilistic rational programming language. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, 2001.

[66] Martyn Plummer and Others. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. *Proceedings of the 3rd international workshop on distributed statistical computing*, 2003.

[67] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. *Proceedings of the IEEE International Conference on Computer Vision*, pages 689–690, 2011.

[68] Bob Rehder and ShinWoo Kim. How causal knowledge affects classification: A generative theory of categorization. *Journal of experimental psychology. Learning, memory, and cognition*, 32:659–683, January 2006. ISSN 0278-7393.

[69] Gareth O. Roberts and Jeffrey S. Rosenthal. Examples of adaptive MCMC. *Journal of Computational and Graphical Statistics*, 18(2):349–367, 2009. ISSN 1061-8600.

[70] Antonio Salmeron, Andrés Cano, and Serafin Moral. Importance sampling in Bayesian networks using probability trees. *Computational Statistics and Data Analysis*, 34(4):387–413, October 2000.

[71] Adam N Sanborn, Vikash K Mansinghka, and Thomas L Griffiths. Reconciling intuitive physics and Newtonian mechanics for colliding objects. *Psychological Review*, 120(2):411, April 2013.

[72] Taisuke Sato. Generative Modeling by PRISM. In Patricia Hill and David Warren, editors, *Logic Programming*, volume 5649 of *Lecture Notes in Computer Science*, chapter 4, pages 24–35. Springer, Berlin, Heidelberg, 2009. ISBN 978-3-642-02845-8.

[73] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1-3):7–42, 2002.

[74] Thomas C Schelling. *The Strategy of Conflict*. Harvard University Press, 1960.

[75] Laura Schulz. Finding New Facts; Thinking New Thoughts. *Advances in Child Development and Behavior*, 43:269–294, 2012.

[76] Ross D Shachter and Mark A Peot. Simulation Approaches to General Probabilistic Inference on Belief Networks. In *Proc. of the 5th Ann. Conf. on Uncertainty in A.I. (UAI-89)*, pages 311–318, New York, NY, 1989. Elsevier Science.

[77] Patrick Shafto, Noah D. Goodman, and Thomas L. Griffiths. A rational account of pedagogical reasoning: Teaching by, and learning from, examples. *Cognitive Psychology*, 71:55–89, 2014. ISSN 00100285.

[78] Lei Shi, Naomi H Feldman, and Thomas L Griffiths. Performing Bayesian inference with exemplar models. *Proceedings of the 30th Annual Conference of the Cognitive Science Society*, pages 745–750, 2008.

[79] Stuart M Shieber, Yves Schabes, and Fernando C N Pereira. Principles and implementation of deductive parsing. *The Journal of Logic Programming*, 24 (1-2):3–36, 1995.

[80] Jacob Steinhardt and Percy Liang. Filtering with Abstract Particles. In *Proceedings of The Thirty-First International Conference on Machine Learning*, pages 727–735, June 2014.

[81] Andreas Stolcke and Stephen Omohundro. Inducing probabilistic grammars by bayesian model merging. In *Grammatical inference and applications*, pages 106–118. Springer, 1994.

[82] Andreas Stuhlmüller and Noah D Goodman. A Dynamic Programming Algorithm for Inference in Recursive Probabilistic Programs. *Second Statistical Relational AI workshop at UAI 2012 (StaRAI-12)*, 2012.

[83] Andreas Stuhlmüller and Noah D Goodman. Reasoning about Reasoning by Nested Conditioning: Modeling Theory of Mind with Probabilistic Programs. *Cognitive Systems Research*, 2013. ISSN 1389-0417.

[84] Andreas Stuhlmüller, Joshua B Tenenbaum, and Noah D Goodman. Learning Structured Generative Concepts. In *Proceedings of the Thirty-Second Annual Conference of the Cognitive Science Society*, 2010.

[85] Andreas Stuhlmüller, Jessica Taylor, and Noah D Goodman. Learning Stochastic Inverses. *Advances in Neural Information Processing Systems (NIPS) 27*, 2013.

[86] Andreas Stuhlmüller, Robert X.D. Hawkins, Siddharth N, and Noah D. Goodman. Coarse-to-Fine Sequential Monte Carlo for Probabilistic Programs. In prep.

[87] Richard Szeliski, Ramin Zabih, Daniel Scharstein, Olga Veksler, Vladimir Kolmogorov, Aseem Agarwala, Marshall Tappen, and Carsten Rother. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(6):1068–1080, 2008.

[88] Yun Tang, Wen-Ju Liu, Hua Zhang, Bo Xu, and Guo-Hong Ding. One-pass coarse-to-fine segmental speech decoding algorithm. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 1, pages I–I. IEEE, 2006.

[89] Terence Tao. *Poincaré's Legacies*. Pages from Year Two of a Mathematical Blog. Amer Mathematical Society, 2009.

[90] Terry Tao. The blue-eyed islanders puzzle, 2008.

[91] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[92] Joshua B Tenenbaum, Charles Kemp, Thomas L Griffiths, and Noah D Goodman. How to Grow a Mind: Statistics, Structure, and Abstraction. *Science*, 331(6022):1279–1285, 2011.

[93] Marc T Tomlinson and Bradley C Love. From Pigeons to Humans : Grounding Relational Learning in Concrete Examples. *AAAI*, 21(1):199–204, 2006.

[94] Marc Toussaint, Stefan Harmeling, and Amos Storkey. Probabilistic inference for solving (PO)MDPs. *Institute for Adaptive and Neural Computation*, 2006.

[95] Katsumi Watanabe and Shinsuke Shimojo. When sound affects vision: effects of auditory grouping on visual motion perception. *Psychological Science*, 12 (2):109–116, 2001.

[96] Henry M Wellman. *The child's theory of mind*. The MIT Press, October 1992.

[97] Kenneth G Wilson. The renormalization group: Critical phenomena and the Kondo problem. *Reviews of Modern Physics*, 47(4):773, 1975.

[98] Kenneth G. Wilson. Problems in Physics with many Scales of Length. *Scientific American*, 241:158–179, 1979. ISSN 0036-8733.

[99] David Wingate and Theophane Weber. Automated Variational Inference in Probabilistic Programming. *arXiv preprint arXiv:1301.1299*, 2013.

[100] David Wingate, Andreas Stuhlmüller, and Noah D Goodman. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, 2011.

[101] David Wingate, Andreas Stuhlmüller, and Noah D Goodman. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, pages 1–9, March 2011.

[102] Haohai Yu and Robert A Van Engelen. Refractor importance sampling. *arXiv preprint arXiv:1206.3295*, 2012.

[103] Changhe Yuan and Marek J Druzdzel. Importance sampling in Bayesian networks: An influence-based approximation strategy for importance functions. *arXiv preprint arXiv:1207.1422*, 2012.