Probabilistic Models with Unknown Objects

by

Brian Christopher Milch

B.S. (Stanford University) 2000

A dissertation submitted in partial satisfaction

of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

and the Designated Emphasis

in

Communication, Computation and Statistics

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Stuart J. Russell, Chair
Professor Michael I. Jordan
Professor Dan Klein
Professor James Pitman

Fall 2006

The dissertation of Brian Christopher Milch is approved:

Chair _____ Date _____
Professor Stuart J. Russell

_____ Date _____
Professor Michael I. Jordan

_____ Date _____
Professor Dan Klein

_____ Date _____
Professor James Pitman

University of California, Berkeley

Fall 2006

Probabilistic Models with Unknown Objects

by

Brian Christopher Milch

Abstract

Probabilistic Models with Unknown Objects

by

Brian Christopher Milch

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Stuart J. Russell, Chair

Humans and other intelligent agents must make inferences about the real-world objects that underlie their observations: for instance, the objects visible in an image, or the people mentioned in a set of text documents. The agent may not know in advance how many objects exist, how they are related to each other, or which observations correspond to which underlying objects. Existing declarative representations for probabilistic models do not capture the structure of such scenarios.

This thesis introduces Bayesian logic (BLOG), a first-order probabilistic modeling language that specifies probability distributions over possible worlds with varying sets of objects. A BLOG model contains statements that define conditional probability distributions for a certain set of random variables; the model also specifies certain context-specific independence properties. We provide criteria under which such a model is guaranteed to fully define a probability distribution. These criteria go beyond existing results in that they can be satisfied even when the Bayesian network defined by the model is cyclic, or contains nodes with infinitely many ancestors.

We describe several approximate inference algorithms that exploit the context-specific dependence structure revealed by a BLOG model. First, we present rejection sampling and likelihood weighting algorithms that are guaranteed to converge to the

correct probability for any query on a structurally well-defined BLOG model. Because these algorithms instantiate only those variables that are context-specifically relevant, they can generate samples in finite time even when the model defines infinitely many variables. We then define a general framework for inference on BLOG models using Markov chain Monte Carlo (MCMC) algorithms. This framework allows a programmer to plug in a domain-specific proposal distribution, which helps the Markov chain move to high-probability worlds. Furthermore, the chain can operate on partial world descriptions that specify values only for context-specifically relevant variables. We give conditions under which MCMC over such partial world descriptions is guaranteed to converge to correct probabilities. We also show that this framework performs efficiently on a real-world task: reconstructing the set of distinct publications referred to by a set of bibliographic citations.

Professor Stuart J. Russell, Chair Date

Acknowledgements

I could not have completed this thesis without the guidance and support of many teachers, colleagues, and friends. The first thanks go to my Ph.D. advisor, Stuart Russell. Without fail, Stuart has encouraged me to keep my eye on the big problems of our field; to combine elegant theory with concrete implementation; and to present my work to a variety of audiences with the greatest possible clarity and precision. I am grateful to him for his patience and his confidence in what I could accomplish. I would also like to thank my undergraduate advisor, Daphne Koller, who welcomed me into her research group and set me on the path to a career in artificial intelligence. In Daphne's group, I also benefited from the mentorship of Avi Pfeffer, whose insight and generosity remain an inspiration to me.

The members of my thesis committee also deserve special thanks for contributing their time and advice. Michael Jordan has provided especially valuable comments, and his courses on machine learning were among the most important parts of my graduate education. I am grateful to Dan Klein for useful conversations about research, as well as about teaching and other aspects of academic life. James Pitman's course on probability theory vastly deepened my understanding of the topic, and he has also been a responsive and helpful member of my committee.

Much of the work described in this thesis was done jointly with my fellow graduate student Bhaskara Marthi. Working with Bhaskara over the past five years has been a great pleasure and privilege; his deep technical background and insights have smoothed the way through many rough patches. I have also had the honor of working with three outstanding undergraduates, Andrey Kolobov, Daniel Ong, and David Sontag, who made important contributions to several aspects of the project and have become valued collaborators and friends. I am grateful as well to Hanna Pasula, who

initiated our group's work on first-order probabilistic languages and helped me get started in research at Berkeley.

Many other AI graduate students and post-docs at Berkeley gave feedback on my talks, helped me work through ideas, and provided support and companionship. I would especially like to thank Norm Aleks, Eyal Amir, Barbara Engelhardt, Greg Lawrence, XuanLong Nguyen, Mark Paskin, Jason Wolfe, and Eric Xing.

I am thankful to have received financial support during graduate school from a University of California MICRO Fellowship, a National Science Foundation Graduate Research Fellowship, and a Siebel Scholarship. My research has also been supported by the Office of Naval Research under MURI N00014-00-1-0637, and by the Defense Advanced Research Projects Agency (DARPA) under contracts 03-000219 and FA 8750-05-2-0249.

Man does not live by research alone; I could not have persevered through this process without the social support of the close friends I made at both Stanford and Berkeley. I am grateful to them for keeping me going and helping me unwind.

My last note of thanks is to my parents. From day one, they encouraged my intellectual curiosity and my desire to figure things out. They have been my best cheerleaders through graduate school, while giving me an unconditional net of support to fall back on. Mom and Dad, thank you.

To my grandparents, one of whom memorably referred to me as "the future Dr. Milch" when I was at an impressionable age.

Contents

1 Introduction **1**

1.1 Declarative probabilistic models 2

1.2 Models with unknown objects 6

1.3 Overview of the thesis 10

2 Background **15**

2.1 Logic 15

2.2 Probability 24

2.3 Graphs and numberings 36

2.4 Bayesian networks 42

2.5 Sampling methods for probabilistic inference 55

3 Contingent Probabilistic Models **69**

3.1 Motivation 69

3.2 Non-product outcome spaces 75

3.3 Split trees 80

3.4 Partition-based models 89

3.5 Contingent Bayesian networks 114

Appendix 3.A Another factorization property 128

Appendix 3.B Supportive split trees revisited 129

4 Bayesian Logic (BLOG) 135

4.1 Examples . . . 138

4.2 Syntax . . . 145

4.3 Declarative semantics . . . 168

4.4 Evaluating expressions . . . 196

4.5 Structurally well-defined BLOG models . . . 229

Appendix 4.A Measurability of BLOG expressions . . . 246

5 Inference for BLOG Models 251

5.1 Evidence and queries . . . 252

5.2 Rejection sampling . . . 261

5.3 Likelihood weighting . . . 274

5.4 Markov chain Monte Carlo . . . 285

5.5 Application to citation matching . . . 299

6 Related Work 321

6.1 Contingent dependencies . . . 321

6.2 Infinite models . . . 322

6.3 First-order probabilistic languages . . . 324

7 Conclusion 337

7.1 Contributions of this thesis . . . 337

7.2 Directions for future research . . . 341

Bibliography 347

Chapter 1

Introduction

A central aspect of intelligence is the ability to make inferences about the real-world objects that underlie one's observations. As human beings, we do this all the time, often subconsciously. For instance, suppose I arrive in a new office and find a computer monitor on my desk. I turn it on and discover, to my disappointment, that it does not work. When I come in the next day, I have every reason to believe that the monitor on my desk is the same one I saw yesterday, and still does not work. On the other hand, suppose I arrive on the third day and find a note on my desk saying, "Broken equipment has been replaced. Regards, Tech Support". At this point I can infer that the phrase "broken equipment" in the note refers to the old monitor, and that the monitor on my desk today probably works — even if it looks identical to the old one.

Behaving intelligently in such a situation requires determining when an image that I'm seeing, or a phrase that I'm reading, corresponds to an object that I've seen before. Figure 1.1 illustrates the conclusions that I reach in the example above: all my observations correspond to just two monitors, the old one and the new one. Making these inferences does not just involve mapping my observations to a pre-

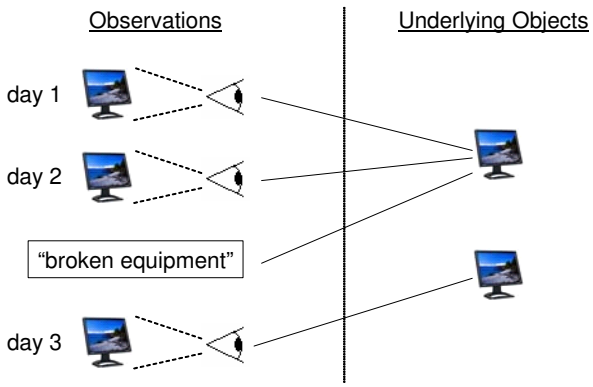


Figure 1.1: Illustration of the computer monitor scenario.

specified list of underlying objects — after all, I was not born with a list of all the individual objects I would ever need to reason about. Instead, I routinely hypothesize new objects as I get new observations. Furthermore, my inferences about what I'm seeing or reading about are not based on some infallible set of rules. Someone could have replaced the monitor before my second day of work without leaving a note; or the phrase "broken equipment" could refer to some other object in the room. My inferences are uncertain: I adopt some hypotheses because I consider them more likely than the alternatives given my observations so far.

If we want to build robots that can perform useful tasks in real-world environments, or web search engines that can answer questions about people, places and things rather than just finding words and phrases, we will need to automate this kind of inference. This thesis takes a step in that direction.

1.1 Declarative probabilistic models

Probabilistic inference must be based on some knowledge of the relative likelihoods of various courses of events, or "possible worlds". Such knowledge is represented mathematically by a *probabilistic model*. A probabilistic model for the broken monitor

scenario would have to specify, for example, the probability that a monitor would be replaced overnight; the probability that someone replacing a monitor would leave a note; the probability that a new monitor would look identical to the old one; and the probability that there would be some other broken equipment in an office besides the monitor. This is an incomplete list of probabilities: a complete probabilistic model for this scenario would be quite complicated.

There are, by and large, two ways to build a computer system that uses a probabilistic model. In the first approach, the engineers define the probability model on paper, work out an algorithm for performing inference on this model, and then write code that implements the algorithm for this particular model. It is often possible to adjust the numerical parameters of the model (perhaps estimating them from data) without changing the code. However, more significant changes to the model often require rewriting parts of the software.

This approach is unsatisfying for several reasons. First, it requires a lot of engineering effort for each new model that is developed. Second, it can be unclear what probability model a system is actually using; thus, it can be difficult to spot errors in the code, and to tell whether a more efficient inference algorithm could be applied to the same model. Finally, this approach provides little insight into how a general-purpose reasoning system — such as the human brain — might operate. Humans seem to be able to learn new probabilistic models from experience, and then do inference on these models without having an engineer implant new inference algorithms.

The alternative approach is to use *declarative* representations of probability models: computer-readable representations that have well-defined semantics in terms of probability distributions over possible worlds, independently of any particular inference algorithm. Once a declarative representation language is established, researchers can develop inference algorithms that work on large classes of models

defined in this language. These algorithms can then be applied, with little or no additional programming, to models from any real-world domain. Algorithms can also be developed to learn models from data.

This declarative approach is also fraught with challenges, however. The number of possible worlds is usually enormous (even if the scenario just consists of n coin tosses, the number of possible worlds is 2^n), so a model cannot simply list the probability of each world. Also, it may be computationally intractable to compute the probability of a query event (such as, "the monitor on my desk today works") given some observations. In general, this involves looking at all the worlds consistent with the observations, and seeing what fraction of their total probability is assigned to worlds where the query event occurs. Special-purpose algorithms can answer queries more efficiently by exploiting structure in particular models, but finding algorithms that exploit structure in a large class of models is difficult.

Nevertheless, the declarative approach to probabilistic reasoning has seen great successes over the past twenty years. The representation formalism responsible for this success is *graphical models* [Pearl, 1988; Cowell *et al.*, 1999], including *Bayesian networks* (BNs), which are based on directed graphs, and *Markov networks*, which are based on undirected graphs. We give a more thorough introduction to (directed) graphical models in Section 2.4, but the basic idea is as follows. It is assumed that the model has been decomposed into a set of random variables, such as Monitor1WorksOnDay2 (which takes a true/false value) and TheMonitorOnMyDeskOnDay2 (which takes a monitor as its value). The graphical part of the model contains one node for each random variable. Edges (links) between the nodes represent probabilistic dependencies: for instance, as shown in Figure 1.2, TheMonitorOnMyDeskOnDay2 depends on TheMonitorOnMyDeskOnDay1 and MonitorReplacedAfterDay1. The model is also equipped with numerical parameters that define exactly how each variable depends on its neighbors. This representa-

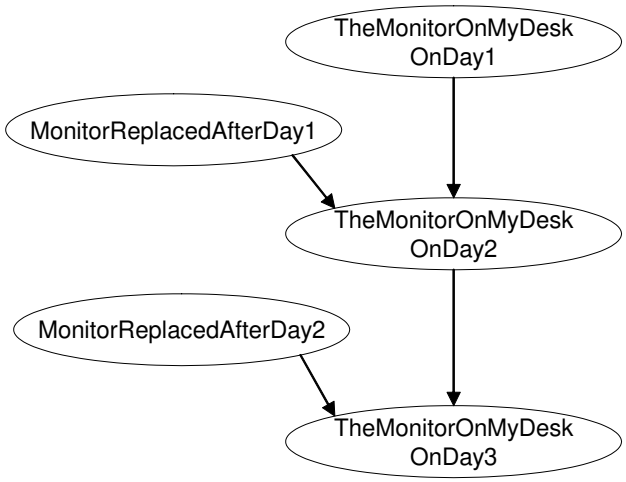


Figure 1.2: A Bayesian network describing how the monitor on my desk on a given day depends on which monitor was on my desk on the previous day, and on whether the monitor was replaced.

tion is particularly useful because the missing edges in the graph reveal probabilistic independence properties, which can be exploited by general-purpose inference algorithms.

One limitation of graphical models is that they are propositional, as opposed to first-order: in other words, they do not support quantification over objects. Thus, if we want to reason about M separate monitors, we need to repeat the portions of the graph that deal with monitors M times. However, a number of *first-order probabilistic languages* (FOPLs) have been developed to define graphical models compactly for scenarios with multiple objects [Horsch and Poole, 1990; Gilks *et al.*, 1994; Koller and Pfeffer, 1998]. These languages allow us to define indexed families of random variables: for example, random variables $\mathsf{MonitorWorks}(m, d)$ for each monitor m and day d. We review these languages in more detail in Section 6.3.

In this thesis, we focus on a further shortcoming of conventional graphical models: they do not appropriately capture the structure of scenarios with unknown objects. First, it is not obvious how to take a scenario involving an unknown number of

objects and decompose it into a fixed set of random variables. If we do not know how many monitors we will need to reason about, then how many nodes of the form MonitorWorks(m, Day1) do we include in our graphical model? If we do not wish to place any upper bound on the number of monitors we can reason about, then we need to include an infinite number of these nodes. But the theory of infinite graphical models is not very well developed—and such a model does not represent the fact that some random variables are applicable only when the corresponding objects exist.

Second, if we do not know how observations are related to underlying objects, then the dependency structure of the model becomes *contingent* rather than fixed. For instance, suppose our model includes a variable MonitorOnMyDeskWorks(Day1) that I can observe on day one. This variable depends on the monitor-valued variable TheMonitorOnMyDesk(Day1), and also on *some* variable of the form MonitorWorks(m, Day1) — specifically the one with m = TheMonitorOnMyDesk(Day1). But if we want to capture this dependency in a graphical model with fixed edges, the only thing we can do is include edges into MonitorOnMyDeskWorks(Day1) from *all* the MonitorWorks(m, Day1) variables (if we left out any of those edges, the model would assert an independence property that does not hold in general). In the resulting model, shown in Figure 1.3, the contingent nature of the dependency is not represented in the graph structure; it is only encoded in the numerical parameters.

1.2 Models with unknown objects

Despite these shortcomings of existing declarative representations, systems that reason about unknown objects using probabilistic models have indeed been built and used successfully. These systems have not used general-purpose declarative representations, and have been limited to specific domains.

One prominent application that calls for probabilistic reasoning about unknown

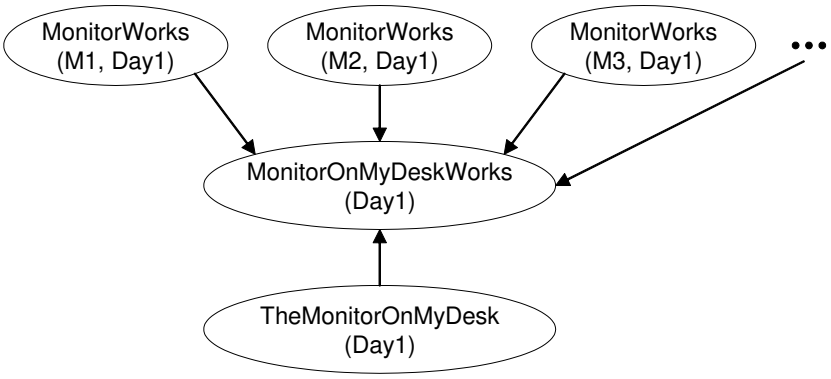


Figure 1.3: A graphical model for a scenario where there is uncertainty about which monitor is on my desk, and there is no *a priori* upper bound on the number of monitors that exist.

objects is the task of tracking multiple objects using cameras or radar. This task is often called *multitarget tracking*, and the problem of determining which observations (radar blips, image segments, etc.) correspond to which underlying objects is called *data association*. The field dates back to early work by Sittler [1964]; there is a standard textbook by Bar-Shalom and Fortmann [1988]. The probabilistic models used for multitarget tracking describe how targets move over time and how observations depend on the true positions of targets; they can also specify probabilities for new targets arriving, targets failing to be observed, and spurious observations appearing at any given time step. If we observe, say, the positions of blips on a radar screen over several time steps, we can use such a model to compute the probability that two particular blips came from the same aircraft.

Performing such computations becomes computationally intractable as the number of observations increases, because we need to consider all possible mappings between observations and underlying objects. Thus, in practice, data association is performed using various heuristic approximation algorithms. However, these algorithms do not operate on a declarative representation of a probabilistic model; they are specific to multi-target tracking. Modifying such algorithms to go beyond the

basic multi-target tracking model — for instance, to exploit the fact that aircraft fly in formation, or to infer the locations of air bases where aircraft often take off and land — often requires a major research effort.

The problem of unknown objects arises not just with direct sensory observations, but also with linguistic input. For instance, if a newspaper article uses the noun phrases, "a police officer", "Robert Smith", "he", and "a suspect", how many distinct people are being referred to? We would have to read the article to figure this out. In computational linguistics, the task of determining which noun phrases refer to the same objects is called *coreference resolution* or *anaphora resolution* [Lappin and Leass, 1994; MUC–6, 1995; Soon *et al.*, 2001]. And even if a computer system gets all its information from databases, rather than physical sensors or raw text, it may still have to determine when a set of database records refer to the same real-world object. If one record is about "Brian Mitch" in "Berkeley, CA" and another is about "Brian C. Milch" in "Berkley, California", are these records about the same person? This problem goes by many names, including *record linkage*, *entity resolution*, and *deduplication*. It has been an active field of study since the 1950s [Newcombe *et al.*, 1959; Fellegi and Sunter, 1969]; a recent report by Winkler [2006] gives a good overview of the field.

Most work in coreference resolution and record linkage has focused on evaluating the probability that a single pair of mentions (or records) refer to the same entity. This probability is based on similarities between the mentions, and also on proximity and syntactic cues when the mentions are noun phrases in a document. Once these probabilities are determined, one can attempt to find the partition of the mentions that best respects all the pairwise match probabilities [McCallum and Wellner, 2005; Singla and Domingos, 2005]. By reasoning only about pairwise matches between mentions, these techniques avoid reasoning explicitly about the unknown objects that the mentions refer to.

However, there are cases where reasoning about the attributes of underlying objects is necessary [Milch *et al.*, 2004]. For instance, suppose a document contains the noun phrases "Alice", "Stuart", and "Jones", and there are syntactic cues suggesting that all three of them co-refer. If we just look at pairwise compatibilities, it seems plausible that the mentions could all be coreferent: there could easily be a person named "Alice Stuart", "Stuart Jones" or "Alice Jones". However, "Alice" and "Stuart" can plausibly co-refer only if "Stuart" is a surname, while "Stuart" and "Jones" can plausibly co-refer only if "Stuart" is a first name. It is not likely that one person would be referred to by all three of these names. Thus, in order to do coreference resolution in general, we need to consider hypotheses about the attributes of underlying objects that might be referred to by the mentions. There are only a handful of coreference resolution algorithms that do this kind of reasoning [Pasula *et al.*, 2003; Wellner *et al.*, 2004]. Interestingly, reasoning about attributes of the underlying objects is standard in the multitarget tracking literature: if one does not represent the velocities of the objects (specifically their direction of travel), then it is difficult to keep track of, say, two objects that pass close to each other while moving in opposite directions. As in the multitarget tracking case, the coreference resolution techniques that reason about unknown objects use special-purpose representations and inference algorithms.

Another application area that involves reasoning about unknown objects is *population estimation*: for instance, estimating the population of a certain animal species in an area by marking animals that are caught in one sweep through the area, and looking at the fraction of marked animals in a subsequent sweep [Borchers *et al.*, 2002]. This application differs from the ones discussed above in that tracking individual objects (animals in this case) is no longer the goal; instead, one is primarily interested in the *number* of unknown objects. Again, this field tends to use specialized techniques rather than general-purpose declarative formalisms.

Finally, work in statistics on *Bayesian mixture models* can be thought of as dealing with unknown objects. In a Bayesian mixture model, observations are generated by first choosing a *mixture component* from some prior distribution, and then choosing values for the observable attributes according to a probability distribution associated with that mixture component. For example, a mixture model for stars might have a component for each type of star (blue giant, yellow dwarf, white dwarf, etc.), each with its own distribution for color, mass, and luminosity. The mixture components are the unknown objects here: the number of components may be fixed [Cheeseman *et al.*, 1988; Neal, 1991], it may be unknown but have some prior distribution [Richardson and Green, 1997], or it may be infinite [Ferguson, 1983; Neal, 1991; Escobar and West, 1995]. If the number of mixture components has an upper bound (or if a model with infinitely many components is truncated, as in the work of Ishwaran and James [2001]), then the model can be represented as a standard Bayesian network. But when the number of components is unbounded or infinite, the only alternatives are to truncate the model or apply a special-purpose inference algorithm.

1.3 Overview of the thesis

The primary contribution of this thesis is a new first-order probabilistic language, called *Bayesian logic* or BLOG, that concisely defines distributions over possible worlds with unknown objects. These possible worlds are represented formally as model structures of a first-order logical language; the set of objects that exist can vary from world to world. BLOG makes explicit the contingent dependencies that arise from relational uncertainty and the unknown mapping from observations to objects. We describe new inference algorithms that exploit this explicit structure to answer queries more efficiently than existing algorithms, and even perform inference

in finite time on some BLOG models that define infinite Bayesian networks. These algorithms have been implemented in a system that is publicly available from the author's web page.

We begin in Chapter 2 with some technical background on first-order logic, probability theory, Bayesian networks, and sampling-based algorithms for approximate inference. The novel material begins in Chapter 3, where we introduce two new declarative formalisms for describing probability models with contingent dependencies. These formalisms are propositional rather than first-order; they form the foundation for BLOG in the same way that standard graphical models form the foundation for existing first-order probabilistic languages. The first formalism, *partition-based models* (PBMs), provides an abstract way of specifying a probabilistic model in terms of conditional distributions for individual variables along with context-specific independence assertions. The second formalism, called *contingent Bayesian networks* (CBNs), provides a concrete way of expressing a PBM. A CBN differs from an ordinary Bayesian network in that the edges are labeled with conditions indicating when they are active. We provide criteria under which a CBN is guaranteed to fully define a probability distribution over outcomes; these criteria go beyond existing results for BNs in that they can be satisfied even if some nodes in the graph have infinitely many ancestors, and even if the graph contains cycles.

Chapter 4 defines the syntax and semantics of the BLOG language. Intuitively, a BLOG model defines a generative process that constructs a possible world step by step, where each step either sets the value of a function on some arguments, or adds some new objects to the world. More formally, each BLOG model defines a PBM over a certain set of basic random variables. We show that a probability distribution for these basic random variables always corresponds uniquely to a probability distribution over first-order model structures. Thus, building on results from Chapter 3, we show that every BLOG model in a large class of *structurally well-defined* models

fully defines a distribution over possible worlds.

In Chapter 5, we move on to the question of inference in BLOG models. We begin by describing *rejection sampling* and *likelihood weighting* algorithms that are guaranteed to converge to the correct probability for any query on a structurally well-defined BLOG model. These algorithms exploit the contingent dependency structure revealed by the BLOG model to avoid sampling irrelevant random variables. In fact, these algorithms can generate samples in finite time even if the BLOG model defines an infinite Bayesian network, as we illustrate with experiments on a pedagogical "balls in an urn" example.

These two algorithms both sample variables top-down according to their prior distributions, and thus they may take astronomical amounts of time to converge on problems of practical interest. In Section 5.4, however, we describe a framework for doing inference on BLOG models using *Markov chain Monte Carlo* (MCMC) algorithms. An MCMC algorithm performs a random walk over possible worlds in such a way that in the long run, the number of times each world is visited is proportional to its probability under the model. Our framework allows a programmer to provide a domain-specific *proposal distribution*, which helps the MCMC process to move between high-probability worlds more efficiently. The main innovation in our framework is that the chain does not have to run over fully specified possible worlds: instead, the MCMC states can be *partial* world descriptions that only specify values for relevant variables. Again, the relevant variables are determined context-specifically, based on the model's contingent dependency structure. We provide conditions under which this abstract-level MCMC algorithm is guaranteed to converge to correct probabilities.

Finally, in Section 5.5, we describe an application of BLOG to a real-world task: parsing the citations that are found at the ends of academic papers, and constructing a de-duplicated database of the publications that they refer to. We discuss a BLOG

model that we developed for this task, as well as a custom proposal distribution that can be plugged into our general MCMC inference engine. Our experimental results show that even though the BLOG inference engine is implemented in a general-purpose way — to handle arbitrary BLOG models and proposal distributions — its speed is within a reasonable constant factor of a hand-coded, domain-specific implementation.

Chapter 6 discusses how this thesis relates to existing work: particularly, how BLOG compares with other first-order probabilistic languages. Some of these languages are quite different from BLOG in their approach to defining probability models, and thus it is difficult to compare their expressive power to BLOG's directly. However, in the category of languages that use directed graphical models to define distributions over relational structures, BLOG is the most expressive.

Chapter 7 concludes the thesis with a summary of contributions and some thoughts on future work. The main areas for future research are learning the parameters and structure of BLOG models, and developing general-purpose inference algorithms that can handle more realistic BLOG models.

Much of the material in this thesis has been described in a series of conference papers. PBMs and CBNs are defined in a paper presented at the 2005 AI/Stats Conference [Milch *et al.*, 2005b]. BLOG was first described informally at the ICML 2004 Workshop on Statistical Relational Learning and Its Connections to Other Fields [Milch *et al.*, 2004], and presented formally at the 2005 International Joint Conference on Artificial Intelligence [Milch *et al.*, 2005a]. The MCMC inference framework was described at the 2006 Conference on Uncertainty in AI [Milch and Russell, 2006].

Chapter 2

Background

2.1 Logic

First-order logic plays a foundational role in both the syntax and the semantics of the BLOG language. This section briefly reviews the relevant concepts. It also discusses how we extend standard first-order logic to distinguish between different types of objects, and to allow partial functions. For a thorough introduction to logic, see [Enderton, 2001].

2.1.1 Propositional logic

The reader is probably familiar with propositional logic, where symbols such as p and q are used to represent *atomic propositions* such as "Alice works in Soda Hall" and "Alice has a key to Soda Hall". A *propositional language* for a particular application domain is simply a set of atomic proposition symbols. These symbols can be combined to form sentences using the *logical connectives* $\neg$ ("not"), $\wedge$ ("and"), $\vee$ ("or") and $\rightarrow$ ("implies"), as well as parentheses. For instance, $p \wedge q$ might mean, "Alice works in Soda Hall and Alice has a key to Soda Hall". A *truth assignment*

is a function that maps each atomic proposition symbol in a language to a value in $\{\mathsf{true}, \mathsf{false}\}$. A truth assignment *satisfies* certain sentences and not others; for instance, $p \wedge q$ is satisfied just by those truth assignments that map both p and q to true.

The drawback of propositional logic is that it is an extremely inefficient way to represent knowledge. For instance, using the symbols defined above, we can write $p \rightarrow q$ to mean "If Alice works in Soda Hall, then Alice has a key to Soda Hall". But this sentence only expresses knowledge about one person, namely Alice, and one building, Soda Hall. If we want to express knowledge of the form, "If person A works in building B, then A has a key to B" for 100 people and 100 buildings, we need 10,000 sentences.

2.1.2 First-order languages

First-order logic allows us to represent knowledge more efficiently by generalizing over objects. A first-order language consists of *constant symbols* such as Alice and $\mathsf{SodaHall}$, *predicate symbols* such as $\mathsf{WorksIn}$ and $\mathsf{HasKeyTo}$, and *function symbols* such as Name and $\mathsf{JobTitle}$. Sentences in first-order logic can contain not only logical connectives, but also the *quantifiers* $\forall$ ("for all") and $\exists$ ("there exists"). We can express our desired statement about people and buildings as $\forall a \, \forall b (\mathsf{WorksIn}(a, b) \rightarrow \mathsf{HasKeyTo}(a, b))$. Here a and b are *logical variables* that range over all objects.[1]

In this thesis we use *typed* (or *sorted*) first-order languages, where the objects are divided into types, and predicate and function symbols apply only to objects of specific types. In our example domain, it is natural to have separate types Person and $\mathsf{Building}$. Instead of using unrestricted quantifiers such as $\forall a$, we use quantifiers that range over a specific type, such as $\forall \ \mathsf{Person} \ a$. Each predicate and function

[1]This kind of logic is called "first-order" to distinguish it from *second-order logic*, where it is possible to quantify over *sets* of objects. Logics of even higher order have also been considered.

symbol has a *type signature*, which specifies the types of arguments it takes and the type of value it returns. In fact, given that we are using a typed language, we can simplify our definitions by treating predicates simply as functions whose return type is Boolean. We will also treat constant symbols as function symbols that take no arguments, that is, *zero-ary* function symbols.

Definition 2.1. *A* typed first-order language $\mathcal{L}$ *is a tuple* $(T_\mathcal{L}, F_\mathcal{L}, \text{sig}_\mathcal{L}, V_\mathcal{L})$ *where:*

- $T_\mathcal{L}$ *is a countable (*i.e.*, finite or countably infinite) set of type symbols, including the Boolean type symbol* Boolean*;*
- $F_\mathcal{L}$ *is a countable set of function symbols;*
- $\text{sig}_\mathcal{L}$ *maps each function symbol* f *to a type signature* $(a_1, \ldots, a_k, r)$*, where* $a_1, \ldots, a_k \in T_\mathcal{L}$ *are* f*'s* argument types *(*$k \geq 0$*), and* $r \in T_\mathcal{L}$ *is* f*'s* return type*;*
- $V_\mathcal{L}$ *is a countable set of logical variable symbols.*

We will also write $\text{ret}_\mathcal{L}(f)$ to denote the return type of a function f. The subscript $\mathcal{L}$ may be omitted when it is clear what language we are talking about.

2.1.3 First-order terms and formulas

The symbols in a first-order language can be put together to form *terms*, which denote objects, and *sentences*, which have truth values.

Definition 2.2. *A* term *in a typed first-order language* $\mathcal{L}$ *has one of the following forms:*

- *the* null term*,* null*;*
- *a* constant term *c, where* $c \in F_\mathcal{L}$ *and* $\text{sig}_\mathcal{L}(c) = (r)$ *for some type* r*;*

- *a* function application term $f(t_1, \ldots, t_k)$ *for some* $k \geq 0$*, where* $f \in F_{\mathcal{L}}$ *and* $\text{sig}_{\mathcal{L}}(f) = (a_1, \ldots, a_k, r)$ *for some types* $a_1, \ldots, a_k$*;*

- *a* logical variable term $v \in V_{\mathcal{L}}$*.*

The only part of this definition that is not standard is the null term, which denotes a special "undefined" value. Its role will become clearer when we discuss semantics in the next section. Another notable aspect of this definition is that the definition of a function application term does not say anything about the types of the argument terms. This is because the type of a logical variable term cannot be determined in isolation; it depends on the *scope* in which the variable occurs.

Definition 2.3. *In a typed first-order language* $\mathcal{L}$*, a* scope β *is a set of pairs* (v, τ)*, where* $v \in V_{\mathcal{L}}$ *and* $\tau \in T_{\mathcal{L}}$*, such that each logical variable* $v \in V_{\mathcal{L}}$ *occurs in at most one pair in* β*.*

We will write $\text{domain}(\beta)$ for the set of variables v that occur in some pair in β. For any $v \in \text{domain}(\beta)$, we will write $\beta(v)$ for the unique type τ such that $(v, \tau) \in \beta$. The *empty scope* is a scope whose domain is the empty set; it does not assign a type to any variables. If β is a scope, we will write $(\beta; v \mapsto \tau)$ to represent the scope formed by taking β, removing any pair that contains v, and adding the pair (v, τ).

Definition 2.4. *The term* t *is a* well-formed term of type r *in a scope* β *if:*

- t *is a constant term* c *and* $\text{sig}_{\mathcal{L}}(c) = (r)$*; or*

- t *is a function application term* $f(t_1, \ldots, t_k)$*,* $sig_{\mathcal{L}}(f) = (a_1, \ldots, a_k, r)$ *for some types* $a_1, \ldots, a_k$*, and* t_i *is a well-formed term of type* a_i *in* β *for each* $i \in \{1, \ldots, k\}$*; or*

- t *is a variable term* v *such that* $(v, r) \in \beta$*.*

The null term is also a well-formed term in all scopes, but it is not of any type.

Now that we have defined what it means for a term to be well-formed in a scope, we can move on to formulas.

Definition 2.5. *A* well-formed formula *of a typed first-order language $\mathcal{L}$ in a scope β has one of the following forms:*

- *a term t that is well-formed and has type* Boolean *in β, called an* atomic formula*;*
- *$t_1 = t_2$ or $t_1 \neq t_2$, where either:*
  - *t_1 and t_2 are well-formed terms of the same type in β, or*
  - *one of t_1, t_2 is a well-formed term in β and the other is the null term;*
- *$\neg\psi$, where ψ is a well-formed formula in β;*
- *$\psi \wedge \chi$, $\psi \vee \chi$, or $\psi \rightarrow \chi$, where ψ and χ are well-formed formulas in β;*
- *$\forall\ \tau\ v\ \psi$ or $\exists\ \tau\ v\ \psi$, where $\tau \in T_{\mathcal{L}}$, $v \in V_{\mathcal{L}}$, and ψ is a well-formed formula in the scope $(\beta; v \mapsto \tau)$.*

The set of *free variables* in a term or formula is the set of variables that are included in all scopes in which that term or formula is well-formed. For instance, in the formula $\forall \mathsf{Person} p \mathsf{WorksIn}(p, b)$, the set of free variables is $\{b\}$. A formula that is well-formed in the empty scope — that is, that has no free variables — is called a *sentence*.

2.1.4 First-order model structures

So far we have discussed only the syntax of first-order logic. We now move on to semantics: the conditions under which sentences are true or false. In propositional

logic, sentences are evaluated with respect to truth assignments. In first-order logic, they are evaluated with respect to more complicated objects called *model structures* (or just *structures*). A model structure for a language $\mathcal{L}$ maps each type to a set of objects, and each function symbol to a function that relates those objects to each other.

Definition 2.6. *A* model structure ω *for a typed first-order language* $\mathcal{L}$ *is a function that:*

- *maps each type* $\tau \in T_{\mathcal{L}}$ *to a set* $[\tau]^{\omega}$*, called the* extension *of* τ *in* ω*, with* $[\mathsf{Boolean}]^{\omega} = \{\mathsf{true}, \mathsf{false}\}$*;*

- *maps each function symbol* $f \in F_{\mathcal{L}}$ *with* $\mathrm{sig}_{\mathcal{L}}(f) = (a_1, \ldots, a_k, r)$ *to a function* $[f]^{\omega}$ *from* $[a_1]^{\omega} \times \cdots \times [a_k]^{\omega}$ *to* $[r]^{\omega} \cup \{\mathsf{null}\}$*, called the* interpretation *of* f *in* ω*.*

For a zero-ary function symbol f, we will abuse notation slightly and use $[f]^{\omega}$ to denote both the function that serves as the interpretation of f, and the value $[f]^{\omega}\,()$ that this function yields on the empty tuple. Also, note that the symbol null serves double duty here: it is both the null term, and an object that can be the value of an interpretation $[f]^{\omega}$ on some arguments.

As an example, consider a language $\mathcal{L}$ with types $\mathsf{Boolean}$, Person and $\mathsf{Building}$, and with the following functions and type signatures:

$$\begin{aligned}
\mathrm{sig}_{\mathcal{L}}(\mathsf{Alice}) &= (\mathsf{Person}) \\
\mathrm{sig}_{\mathcal{L}}(\mathsf{Bob}) &= (\mathsf{Person}) \\
\mathrm{sig}_{\mathcal{L}}(\mathsf{SodaHall}) &= (\mathsf{Building}) \\
\mathrm{sig}_{\mathcal{L}}(\mathsf{WorksIn}) &= (\mathsf{Person}, \mathsf{Building}, \mathsf{Boolean}) \\
\mathrm{sig}_{\mathcal{L}}(\mathsf{HasKeyTo}) &= (\mathsf{Person}, \mathsf{Building}, \mathsf{Boolean})
\end{aligned}$$

In a particular model structure ω, the extension of Person might be $\{\mathsf{P1}, \mathsf{P2}\}$, and the extension of Building might be $\{\mathsf{B1}\}$. The interpretations of Alice, Bob and SodaHall are all zero-ary functions that yield some value on the empty tuple; suppose $[\mathsf{Alice}]^\omega = \mathsf{P1}$, $[\mathsf{Bob}]^\omega = \mathsf{P2}$, and $[\mathsf{SodaHall}]^\omega = \mathsf{B1}$. The interpretation of WorksIn might map $(\mathsf{P1}, \mathsf{B1})$ to true, but $(\mathsf{P2}, \mathsf{B1})$ to false, meaning that Alice works in Soda Hall but Bob does not.

The constant symbols Alice and Bob could also refer to the same object: $[\mathsf{Alice}]^\omega$ and $[\mathsf{Bob}]^\omega$ could both equal P2. This would leave the object P1 with no constant symbols referring to it, which is also allowed. Another possibility is that $[\mathsf{Alice}]^\omega$ or $[\mathsf{Bob}]^\omega$ could equal null.

In each model structure, each sentence of $\mathcal{L}$ is either satisfied or unsatisfied. To define satisfaction, we begin with the notion of assigning values to logical variables. An *assignment* α is a function on a scope β: that is, it maps each variable-type pair $(v, \tau) \in \beta$ to a value. We will write $\mathrm{domain}(\alpha)$ for the scope that α is defined on. If v is a logical variable that occurs in some pair $(v, \tau) \in \mathrm{domain}(\alpha)$, then by the definition of a scope, there is only one such pair; so for such variables v, we can write $\alpha(v)$ as an abbreviation for $\alpha((v, \tau))$. We will write $(\alpha; (v, \tau) \mapsto o)$ to represent the assignment obtained by taking α, removing any mapping for a pair containing v, and adding the mapping $(v, \tau) \mapsto o$. The assignment α is *valid* in a model structure ω if for each $(v, \tau) \in \mathrm{domain}(\alpha)$, $\alpha(v) \in [\tau]^\omega$.

Definition 2.7. *Let ω be a model structure for $\mathcal{L}$, α be an assignment that is valid in ω, and t be a term of $\mathcal{L}$ that is well-formed in* $\mathrm{domain}(\alpha)$*. Then the* denotation *of t in ω under α, denoted $[t]^\omega_\alpha$, is defined as follows:*

- *if t is the null term, then $[t]^\omega_\alpha = \mathsf{null}$;*
- *if t is a constant term c, then $[t]^\omega_\alpha = [c]^\omega$;*

- *if t is a function application term $f(t_1, \ldots, t_k)$, then:*

$$[t]^\omega_\alpha = \begin{cases} \mathsf{null} & \text{if } [t_i]^\omega_\alpha = \mathsf{null} \text{ for some } i \in \{1, \ldots, k\} \\ [f]^\omega\left([t_1]^\omega_\alpha, \ldots, [t_k]^\omega_\alpha\right) & \text{otherwise} \end{cases}$$

- *if t is a logical variable term v, then $[v]^\omega_\alpha = \alpha(v)$.*

Definition 2.8. *Let ω be a model structure for $\mathcal{L}$, α be an assignment that is valid in ω, and φ be a formula of $\mathcal{L}$ that is well-formed in* $\mathrm{domain}(\alpha)$. *Then ω* satisfies *φ under α, written $\omega \models_\alpha \varphi$, if one of the following cases holds:*

- *φ is an atomic formula t and $[t]^\omega_\alpha = \mathsf{true}$;*
- *φ has the form $t_1 = t_2$ and $[t_1]^\omega_\alpha = [t_2]^\omega_\alpha$, or it has the form $t_1 \neq t_2$ and $[t_1]^\omega_\alpha \neq [t_2]^\omega_\alpha$;*
- *φ has the form $\neg\psi$ and $\omega \not\models_\alpha \psi$;*
- *φ has the form $\psi \wedge \chi$ and $\omega \models_\alpha \psi$ and $\omega \models_\alpha \chi$, or it has the form $\psi \vee \chi$ and $\omega \models_\alpha \neg(\neg\psi \wedge \neg\chi)$, or it has the form $\psi \to \chi$ and $\omega \models_\alpha (\neg\psi \vee \chi)$;*
- *φ has the form $\forall\ \tau\ v\ \psi$ and for every $o \in [\tau]^\omega$, $\omega \models_{(\alpha,(v,\tau)\mapsto o)} \psi$;*
- *φ has the form $\exists\ \tau\ v\ \psi$ and $\omega \models_\alpha \neg(\forall\ \tau\ v\ \neg\psi)$.*

If a formula φ has no free variables, then the assignment is irrelevant, and we will just write $\omega \models \varphi$ or $\omega \not\models \varphi$. Similarly, if a term t has no free variables, we will write $[t]^\omega$ for its denotation in ω.

2.1.5 Isomorphisms between model structures

It is possible for two model structures to represent the same relational structure, just with different objects. That is, there may be an *isomorphism* between two model

structures.

Definition 2.9. *Let ω_1 and ω_2 be two model structures of a typed first-order language $\mathcal{L}$. Let h be a bijection between $\bigcup_{(\tau \in T_\mathcal{L})} [\tau]^{\omega_1}$ and $\bigcup (\tau \in T_\mathcal{L}) [\tau]^{\omega_2}$. Then h is an* isomorphism *between ω_1 and ω_2 if:*

- *for each type τ in $\mathcal{L}$ and each object $o \in [\tau]^{\omega_1}$,*

$$h(o) \in [\tau]^{\omega_2};$$

- *for each function symbol f in $\mathcal{L}$ with type signature $(r, a_1, \ldots, a_k)$ and each tuple of arguments $(o_1, \ldots, o_k) \in [a_1]^{\omega_1} \times \cdots \times [a_k]^{\omega_1}$,*

$$h([f]^{\omega_1}(o_1, \ldots, o_k)) = [f]^{\omega_2}(h(o_1), \ldots, h(o_k)).$$

We say two model structures are *isomorphic* if there exists an isomorphism between them. For instance, in the language that we have been using in our examples, let ω_1 be a model structure with:

$$\begin{aligned}
[\mathsf{Person}]^{\omega_1} &= \{\mathsf{P1}, \mathsf{P2}\} \\
[\mathsf{Building}]^{\omega_1} &= \{\mathsf{B1}\} \\
[\mathsf{Alice}]^{\omega_1} &= \mathsf{P1} \\
[\mathsf{Bob}]^{\omega_1} &= \mathsf{P2} \\
[\mathsf{SodaHall}]^{\omega_1} &= \mathsf{B1} \\
[\mathsf{WorksIn}]^{\omega_1} &= \{(\mathsf{P1}, \mathsf{B1}, \mathsf{true}), (\mathsf{P2}, \mathsf{B1}, \mathsf{false})\} \\
[\mathsf{HasKeyTo}]^{\omega_1} &= \{(\mathsf{P1}, \mathsf{B1}, \mathsf{false}), (\mathsf{P2}, \mathsf{B1}, \mathsf{false})\}
\end{aligned}$$

This model structure is isomorphic to a model structure ω_2 that is the same except

that:

$$\begin{aligned} [\mathsf{Alice}]^{\omega_1} &= \mathsf{P2} \\ [\mathsf{Bob}]^{\omega_1} &= \mathsf{P1} \\ [\mathsf{WorksIn}]^{\omega_1} &= \{(\mathsf{P1}, \mathsf{B1}, \mathsf{false}), (\mathsf{P2}, \mathsf{B1}, \mathsf{true})\} \end{aligned}$$

The isomorphism from ω_1 to ω_2 maps P1 to P2, P2 to P1, and B1 to itself.

2.2 Probability

In this section, we assume the reader already has an intuitive understanding of probability. We briefly review the basic concepts and introduce some more advanced material that will be used in later chapters. For a more extensive treatment of this material, see [Durrett, 1996] or [Billingsley, 1995].

2.2.1 Measurable spaces

Let Ω denote a set of possible worlds or outcomes. Intuitively, a probability distribution P assigns a probability $P(\omega)$ to each outcome $\omega \in \Omega$, and it follows that the probability of any event $A \subseteq \Omega$ is $\sum_{(\omega \in A)} P(\omega)$. However, when Ω is uncountable, the probabilities of events can no longer be derived simply by summing the probabilities of individual outcomes. Uncountable outcome sets obviously arise when we deal with real numbers, but we also get uncountable outcome sets when we have infinite sequences. For example, the set of possible outcomes of an infinite sequence of coin tosses is uncountable. If the tosses are independent and each have probability 0.5 of coming up heads, then the probability of any infinite sequence of heads-tails values is zero. But sets of sequences — for example, the set of sequences where the first toss comes up heads — obviously have nonzero probabilities.

Thus, in general, we need to assign probabilities directly to sets of outcomes. For any given scenario, we define an *event space* $\mathscr{F}$ that is a subset of the power set of Ω. Sets in $\mathscr{F}$ are called *measurable sets*. An event space is required to be a *σ-field*: that is, if $A \in \mathscr{F}$ then $(\Omega \setminus A) \in \mathscr{F}$, and if $A_1, A_2, \ldots$ is a countable sequence of sets in $\mathscr{F}$ then $\bigcup_{i=1}^{\infty} A_i \in \mathscr{F}$ (these conditions imply that $\mathscr{F}$ is also closed under countable intersections). A pair $(\Omega, \mathscr{F})$ where Ω is a non-empty set and $\mathscr{F}$ is a σ-field of subsets of Ω is called a *measurable space*.

Generating a σ-field. If $\mathscr{A}$ is a set of subsets of Ω, we write $\sigma(\mathscr{A})$ for the σ-field *generated by* $\mathscr{A}$: that is, the smallest σ-field on Ω that is a superset of $\mathscr{A}$. Intuitively, $\sigma(\mathscr{A})$ is the set of sets that can be generated by starting with the elements of $\mathscr{A}$ and repeatedly taking complements and countable unions and intersections. If a set $\mathscr{A}$ is such that $\sigma(\mathscr{A}) = \mathscr{F}$, then we say that $\mathscr{A}$ *generates* $\mathscr{F}$, or that $\mathscr{A}$ is a *basis* for $\mathscr{F}$.

Discrete spaces. A measurable space $(\Omega, \mathscr{F})$ is *discrete* if Ω is countable (*i.e.*, finite or countably infinite) and $\mathscr{F}$ is the power set of Ω. In this case, $\mathscr{F}$ is generated by the singleton sets $\{\omega\}$ for $\omega \in \Omega$.

Product spaces. If we have a set of measurable spaces $\{(\Omega_i, \mathscr{F}_i)\}_{i \in I}$ indexed by an index set I, then the *product space* $\times_{(i \in I)}(\Omega_i, \mathscr{F}_i)$ is a measurable space $(\Omega, \mathscr{F})$ defined as follows. The outcome set Ω is the set of functions $\omega : I \to \bigcup_{(i \in I)} \Omega_i$ such that $\omega(i) \in \Omega_i$ for each $i \in I$. We will often think of such an outcome ω as a tuple indexed by I, and write ω_i rather than $\omega(i)$. The event space $\mathscr{F}$ is the σ-field generated by the sets of the form $\{\omega : \omega_i \in A\}$ for $i \in I$ and $A \in \mathscr{F}_i$. By convention, if the index set I is empty, then Ω contains just the empty set, which can be interpreted as a function with an empty domain. The *coordinate random*

variables on $\times_{(i \in I)}(\Omega_i, \mathscr{F}_i)$ are a set of variables $\{X_i\}_{(i \in I)}$ such that $X_i(\omega) \triangleq \omega_i$.

2.2.2 Probability measures

A *probability measure* (or *probability distribution*) on a measurable space $(\Omega, \mathscr{F})$ is a function $P : \mathscr{F} \to [0, 1]$ such that:

- $P(\varnothing) = 0$;
- $P(\Omega) = 1$;
- if $A_1, A_2, \ldots$ are disjoint sets in $\mathscr{F}$, then $P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i)$.

Note that if $(\Omega, \mathscr{F})$ is discrete, then every set $A \in \mathscr{F}$ can be expressed as a countable disjoint union of singleton sets in $\mathscr{F}$: $A = \bigcup_{(\omega \in A)}\{\omega\}$. Thus, a probability measure on $(\Omega, \mathscr{F})$ is fully determined by the probabilities assigned to singleton sets. So we can define a *discrete probability distribution* on Ω as a function $P : \Omega \to [0, 1]$ such that $\sum_{(\omega \in \Omega)} P(\omega) = 1$. Then any discrete probability distribution on Ω defines a probability measure on $(\Omega, \mathscr{F})$.

2.2.3 Random variables

A *random variable* on $(\Omega, \mathscr{F})$ is a *measurable* function $X : \Omega \to S_X$, where S_X is a set of possible values equipped with a σ-field $\mathscr{S}_X$. The value set S_X will also be denoted range (X). The requirement that X be measurable means that the *preimages* of measurable sets must be measurable: that is, for each $A \in \mathscr{S}_X$, $\{\omega : X(\omega) \in A\}$ is in $\mathscr{F}$. We will write $\{X \in A\}$ and $\{X = x\}$ as abbreviations for the events $\{\omega : X(\omega) \in A\}$ and $\{\omega : X(\omega) = x\}$.

As an example, suppose Ω is the set of outcomes of an infinite sequence of coin tosses, so each $\omega \in \Omega$ is a sequence $(c_i)_{i=1}^{\infty}$ where each $c_i \in \{H, T\}$. A natural event

space for Ω is the σ-field generated by the events $\{(c_i)_{i=1}^{\infty} : c_n = H\}$ for $n = 1, 2, \ldots$. Given this event space, we can define random variables X_n for $n = 1, 2, \ldots$ such that $X_n\left((c_i)_{i=1}^{\infty}\right) = c_n$. The range of these random variables is the set $\{H, T\}$, with its power set as its σ-field. We can also define random variables H_n yielding the number of heads in the first n tosses. The range of H_n is $\{0, 1, \ldots, n\}$, again with the power set as the σ-field.

A *distribution* for a random variable X is a probability measure on $(S_X, \mathscr{S}_X)$. Any probability measure P on $(\Omega, \mathscr{F})$ *induces* a distribution for X, which we will denote $P(X)$ or simply P_X, such that $P_X(A) \triangleq P(\{\omega : X(\omega) \in A\})$ for each $A \in \mathscr{S}_X$. The requirement that X be a measurable function on $(\mathscr{F}, \Omega)$ ensures that $P(\{\omega : X(\omega) \in A\})$ is well-defined. For a brief proof that the function P_X defined this way is indeed a probability measure on $(S_X, \mathscr{S}_X)$, see Billingsley [1995], pages 185–186. We will sometimes call P_X the *marginal distribution* for X under P. Note that if X is discrete — that is, its value space $(S_X, \mathscr{S}_X)$ is discrete — then P_X can be represented as a function on individual elements of S_X (rather than events).

The idea of an induced distribution can be extended to sets of random variables. We will represent such sets with bold letters, such as $\mathbf{X}$. If $\mathbf{X}$ is a set of random variables on $(\Omega, \mathscr{F})$, then their joint value space is $\text{range}(\mathbf{X}) \triangleq (S_{\mathbf{X}}, \mathscr{S}_{\mathbf{X}}) \triangleq \times_{(X \in \mathbf{X})}(S_X, \mathscr{S}_X)$. We can actually regard $\mathbf{X}$ as one big random variable on $(\Omega, \mathscr{F})$: thus, $\mathbf{X}(\omega)$ is that element σ of $S_{\mathbf{X}}$ such that $\sigma_X = X(\omega)$ for each $X \in \mathbf{X}$. The definition of the σ-field for a product space (see Section 2.2.1) ensures that the function $\mathbf{X}$ defined this way is measurable. It follows that any probability measure P on $(\Omega, \mathscr{F})$ induces a *joint distribution* for $\mathbf{X}$, denoted $P(\mathbf{X})$ or $P_{\mathbf{X}}$, such that $P_{\mathbf{X}}(A) \triangleq P(\{\omega : \mathbf{X}(\omega) \in A\})$ for each $A \in \mathscr{S}_{\mathbf{X}}$.

2.2.4 Instantiations

An *instantiation* of a set of variables $\mathbf{X}$ is an assignment of values to the variables in $\mathbf{X}$. More formally, an instantiation of $\mathbf{X}$ is a function $\sigma : \mathbf{X} \to \bigcup_{(X \in \mathbf{X})} S_X$ such that $\sigma(X) \in S_X$ for each $X \in \mathbf{X}$. Another way to say this is that σ is an element of the product space $\text{range}(\mathbf{X}) \triangleq \times_{(X \in \mathbf{X})}(S_X, \mathscr{S}_X)$. Continuing our convention for product spaces, we will write σ_X instead of $\sigma(X)$ for the value that σ assigns to X.

We will write $\text{vars}(\sigma)$ for the set of variables to which σ assigns a value. Thus, σ is an instantiation of $\text{vars}(\sigma)$. If $\text{vars}(\sigma)$ is a subset of some given set of variables $\mathcal{V}$, then will say that σ is an instantiation *on* $\mathcal{V}$; this use of the preposition "on" is motivated by the fact that such an instantiation is a partial function on $\mathcal{V}$. An instantiation σ on $\mathcal{V}$ is *finite* if $\text{vars}(\sigma)$ is finite, and *complete* if $\text{vars}(\sigma) = \mathcal{V}$.

Each outcome $\omega \in \Omega$ corresponds to a unique instantiation of a set of variables $\mathbf{X}$: the instantiation σ such that $\sigma_X = X(\omega)$ for each $X \in \mathbf{X}$. Conversely, an instantiation σ corresponds to an event $\text{ev}(\sigma) \triangleq \{\omega \in \Omega : X(\omega) = \sigma_X \text{ for all } X \in \text{vars}(\sigma)\}$. We will often just write σ instead of $\text{ev}(\sigma)$ when no confusion is likely: for instance, we will write $P(\sigma)$ instead of $P(\text{ev}(\sigma))$. An outcome ω is *consistent* with σ if $\omega \in \text{ev}(\sigma)$, or in other words, $X(\omega) = \sigma_X$ for all $X \in \text{vars}(\sigma)$. Note that an instantiation may not be consistent with any outcome, in which case its corresponding event is the empty set.

Instantiations play a major role in this thesis, so we introduce quite a bit of terminology and notation for talking about them. We will write $\sigma[\mathbf{Y}]$ for the restriction of σ to a set of variables $\mathbf{Y}$, and σ_{-Y} for the restriction of σ to $\text{vars}(\sigma) \setminus Y$. These are examples of *sub-instantiations* of σ: instantiations τ such that $\text{vars}(\tau) \subseteq \text{vars}(\sigma)$ and $\tau_X = \sigma_X$ for $X \in \text{vars}(\tau)$. Note that if τ is a sub-instantiation of σ, then σ asserts at least as much as τ does, so $\text{ev}(\sigma) \subseteq \text{ev}(\tau)$. Conversely, an *extension* of σ is an instantiation that has σ as a sub-instantiation. The *empty instantiation* is

denoted $\top$; it has the properties $\text{vars}(\top) = \varnothing$ and $\text{ev}(\top) = \Omega$.

Two instantiations σ and τ are *contradictory* if there is some variable $X \in \text{vars}(\sigma) \cap \text{vars}(\tau)$ such that $\sigma_X \neq \tau_X$. If σ and τ are not contradictory, then we will write $(\sigma;\tau)$ for the conjunction of σ and τ. Thus, $\text{ev}(\sigma;\tau) = \text{ev}(\sigma) \cap \text{ev}(\tau)$. If σ and τ are contradictory, we will still allow ourselves to use the expressions $\text{ev}(\sigma;\tau)$ and $P(\sigma;\tau)$: in this case, we interpret $\text{ev}(\sigma;\tau)$ as $\varnothing$, so $P(\sigma;\tau) = 0$. Finally, if $\sigma_1, \sigma_2, \ldots$ is a sequence of non-contradictory instantiations, we will write $\wedge_i \sigma_i$ to denote the conjunction of all these instantiations. That is, $\text{vars}(\wedge_i \sigma_i) = \bigcup_{i=1}^{\infty} \text{vars}(\sigma_i)$, and for every $X \in \text{vars}(\wedge_i \sigma_i)$, the value that $\wedge_i \sigma_i$ assigns to X is that x such that $(\sigma_i)_X = x$ for some i. This x is unique because the instantiations σ_i are non-contradictory.

2.2.5 Defining probability measures

Suppose we specify some constraints on a probability measure over a given measurable space. It is natural to ask under what circumstances there exists a probability measure satisfying these constraints, and under what circumstances this probability measure is unique. The uniqueness question is addressed by the following theorem:

Theorem 2.1 (Theorem 3.3 of [Billingsley, 1995]). *Suppose $\mathscr{A}$ is a set of subsets of Ω that is closed under intersection (that is, if $A \in \mathscr{A}$ and $B \in \mathscr{A}$ then $A \cap B \in \mathscr{A}$) and that generates $\mathscr{F}$. Then any two probability measures that agree on $\mathscr{A}$ agree on all elements of $\mathscr{F}$.*

Thus, if we specify the probabilities of all the events in a set $\mathscr{A}$ satisfying the conditions of this theorem, then there is at most one probability measure satisfying these specifications.

As to the existence of probability measures, we will limit ourselves to outcome spaces formed by taking a countable product of countable spaces. Suppose we are given a countable index set T and a set of measurable spaces $\{(S_t, \mathscr{S}_t)\}_{t \in T}$. Then

we can form the product space $\times_{(t \in T)}(S_t, \mathscr{S}_t)$, which has coordinate random variables $\{X_t\}_{(t \in T)}$ such that $X_t(\omega) \triangleq \omega_t$. The following theorem (a discrete version of Kolmogorov's extension theorem[2]) states that if we specify the probabilities of finite instantiations of the coordinate random variables in a consistent way, then there is a probability measure on the product space that satisfies these probability specifications.

Theorem 2.2 (Theorem A.7.1 in [Durrett, 1996]). *Let T be a countable index set, and let $(\Omega, \mathscr{F}) = \times_{(t \in T)}(S_t, \mathscr{S}_t)$ where each $(S_t, \mathscr{S}_t)$ is a discrete measurable space. Let $X_0, X_1, \ldots$ be any numbering of the coordinate variables on this product space. For each natural number $n \leq |T|$, let f_n be a function from the range space* $\text{range}(\{X_0, \ldots, X_{n-1}\})$ *to $[0, 1]$. Suppose that $f_0(\top) = 1$, and for all natural numbers $n < |T|$ and all instantiations $\sigma \in \text{range}(\{X_0, \ldots, X_{n-1}\})$,*

$$\sum_{x_n \in \text{range}(X_n)} f_{n+1}(\sigma; X_n = x_n) = f_n(\sigma) \tag{2.1}$$

Then there is a unique probability measure P on $(\Omega, \mathscr{F})$ such that $P(\sigma) = f_n(\sigma)$ for each natural number $n \leq |T|$ and each $\sigma \in \text{range}(\{X_0, \ldots, X_{n-1}\})$.

Note that the uniqueness part of this theorem actually follows from Theorem 2.1, with $\mathscr{A}$ consisting of the empty set and all instantiations σ for which $\text{vars}(\sigma) = \{X_0, \ldots, X_{n-1}\}$ for some n.

In some cases, it is also possible to define a probability measure P on $(\Omega, \mathscr{F})$ by defining a probability measure Q on the value space $(S_X, \mathscr{S}_X)$ of a random variable X, and then defining P as the unique probability measure on $(\Omega, \mathscr{F})$ that induces Q. Obviously, the random variable X must have some special properties in order

[2]Kolmogorov's theorem is usually stated with the assumption that the component spaces in the infinite product are all $(\mathbb{R}, \mathscr{B})$. However, the theorem still holds if we assume all the component spaces are *standard Borel spaces*, a class that includes all discrete spaces and all Borel subsets of $\mathbb{R}^n$ (for any n) with their Borel σ-fields (see [Durrett, 1996], p. 33).

for this P to be well-defined. In general, a random variable X depends only on certain aspects of an outcome ω, so there are many probability measures on $(\Omega, \mathscr{F})$ that yield the measure Q on $(S_X, \mathscr{S}_X)$. On the other hand, if Q assigns positive probability to some elements of S_X that do not actually serve as values of X for any $\omega \in \Omega$, then there is no probability measure on $(\Omega, \mathscr{F})$ that induces Q. The following proposition gives conditions under which Q is induced by exactly one probability measure.

Proposition 2.3. *Let $(\Omega, \mathscr{F})$ be a measurable space, let X be a random variable on $(\Omega, \mathscr{F})$ with value space $(S_X, \mathscr{S}_X)$, and let Q be a probability measure on $(S_X, \mathscr{S}_X)$. Further suppose that both of the following conditions hold:*

i. the preimages of events in $\mathscr{S}_X$, denoted $X^{-1}(B)$ for $B \in \mathscr{S}_X$, generate $\mathscr{F}$;

ii. there is some event $S_X^* \in \mathscr{S}_X$ such that $Q(S_X^*) = 1$ and for all $x \in S_X^*$, $X^{-1}(x) \neq \varnothing$.

Then there is a unique probability measure P on $(\Omega, \mathscr{F})$ such that Q is the distribution of X under P.

Proof. Let S_X^* be an event that satisfies the conditions in (ii), and let $\mathscr{S}_X^*$ be the set of events in $\mathscr{S}_X$ that are subsets of S_X^*. Because $S_X^* \in \mathscr{S}_X$, we know $\mathscr{S}_X^*$ is a σ-field, and thus $(S_X^*, \mathscr{S}_X^*)$ is a measurable space.

By (ii), each value $x \in S_X^*$ has a non-empty preimage $X^{-1}(x)$. So we can define a function $h : (S_X^*, \mathscr{S}_X^*) \to (\Omega, \mathscr{F})$ such that for each $x \in S_X^*$, $h(x)$ is an arbitrary element of $X^{-1}(x)$. We claim that h is measurable. Since the events of the form $X^{-1}(B)$ for $B \in \mathscr{S}_X$ generate $\mathscr{F}$, it suffices to show that the events of the form $h^{-1}(X^{-1}(B))$ are $\mathscr{S}_X^*$-measurable. To see this, first consider individual elements $x \in S_X$. Since the sets $X^{-1}(x)$ are disjoint for distinct x, there cannot be any $x' \neq x$

such that $h(x') \in X^{-1}(x)$. So:

$$h^{-1}(X^{-1}(x)) = \begin{cases} \{x\} & \text{if } x \in S_X^* \\ \varnothing & \text{otherwise} \end{cases}$$

Therefore, for any $B \in \mathscr{S}_X$, we have $h^{-1}(X^{-1}(B)) = B \cap S_X^*$, which is an event in $\mathscr{S}_X^*$.

So we can regard h as a random variable from $(S_X^*, \mathscr{S}_X^*)$ to $(\Omega, \mathscr{F})$. We have a probability measure Q on $(S_X, \mathscr{S}_X)$, and by assumption (ii), we know this measure assigns probability one to S_X^* as well. So as discussed earlier in this section, there is a unique probability measure P on $(\Omega, \mathscr{F})$ such that $P(A) = Q(h^{-1}(A))$ for each $A \in \mathscr{F}$.

We will now check that Q is indeed the distribution of X under this P. Consider any $B \in \mathscr{S}_X$. Because X is measurable, we know $X^{-1}(B)$ is an element of $\mathscr{F}$. So by the way we defined P, $P(X^{-1}(B)) = Q(h^{-1}(X^{-1}(B)))$. As we argued above, $h^{-1}(X^{-1}(B)) = B \cap S_X^*$, so we have $P(X^{-1}(B)) = Q(B \cap S_X^*)$. But since $Q(S_X^*) = 1$, we know $Q(B \cap S_X^*) = Q(B)$. So $P(X^{-1}(B)) = Q(B)$, as desired.

Finally, assume for contradiction that there is another probability measure P' that also induces Q as the distribution for X. By definition, for each $B \in \mathscr{S}_X$, $P'(X^{-1}(B)) = Q(B) = P(X^{-1}(B))$. By assumption (i), the events of the form $X^{-1}(B)$ generate $\mathscr{F}$. And this set of events is clearly closed under intersection, since $X^{-1}(B) \cap X^{-1}(B') = X^{-1}(B \cap B')$. So by Theorem 2.1, P' must be identical to P. □

2.2.6 Conditional probability

If P is a probability measure and B is an event such that $P(B) > 0$, then the *conditional probability* of an event A given B is:

$$P(A|B) \triangleq \frac{P(A \cap B)}{P(B)}$$

If $P(A)$ represents our prior degree of belief that A is true, then $P(A|B)$ represents our posterior belief in A given that we know B. We would also like to speak about our beliefs regarding a random variable X given that we know the values of some other variables $\mathbf{W}$: that is, the *conditional probability distribution* for X given $\mathbf{W}$. Our treatment of conditional probability distributions will assume that all the variables involved are discrete, and that the set of variables we are conditioning on is finite. For a discussion of the complexities that arise in extending these ideas to continuous variables or infinite sets of variables, see Section 4.1 of Durrett [1996].

Definition 2.10. *Let X be a discrete random variable, and let $\mathbf{W}$ be a finite set of discrete random variables. A* conditional probability distribution *(CPD) for X given $\mathbf{W}$ is a function $c : \text{range}(X) \times \text{range}(\mathbf{W}) \to [0, 1]$ such that for each $\sigma \in \text{range}(\mathbf{W})$:*

$$\sum_{x \in \text{range}(X)} c(x, \sigma) = 1$$

Just as a probability measure P on $(\Omega, \mathscr{F})$ induces a distribution $P(X)$ for any random variable X defined on $(\Omega, \mathscr{F})$, it seems that P should also induce a CPD for any variable X given any finite set of variables $\mathbf{W}$. The induced CPD c should be defined in terms of the conditional probabilities of events: $c(x, \sigma) = P(X = x|\sigma)$. However, this conditional probability is only defined when $P(\sigma) > 0$. Thus, a probability measure P does not fully specify the CPD for X given $\mathbf{W}$ when some instantiations of $\mathbf{W}$ have probability zero. We can only speak of *versions* of the conditional

distribution $P(X|\mathbf{W})$.

Definition 2.11. *Let P be a probability measure on a measurable space $(\Omega, \mathscr{F})$. Let X be a discrete random variable defined on $(\Omega, \mathscr{F})$, and let $\mathbf{W}$ be a finite set of such variables. Then a CPD c for X given $\mathbf{W}$ is a* version of $P(X|\mathbf{W})$ *if, for every $\sigma \in \text{range}(\mathbf{W})$ with $P(\sigma) > 0$ and every $x \in \text{range}(X)$:*

$$c(x, \sigma) = P(X = x|\sigma)$$

One consequence of this definition is that if c is a version of $P(X|\mathbf{W})$, then:

$$P(\sigma)c(x, \sigma) = P(\sigma)P(X = x|\sigma)$$

This equation holds because the only case when $c(x, \sigma)$ may not be equal to $P(X = x|\sigma)$ is when $P(\sigma) = 0$, and then both sides of the equation are zero anyway. Furthermore, a simple manipulation of the definition of conditional probability shows that $P(X = x|\sigma)P(\sigma) = P(\sigma; X = x)$. Thus, whenever c is a version of $P(X|\mathbf{W})$, we have:

$$P(\sigma)c(x, \sigma) = P(\sigma; X = x) \tag{2.2}$$

2.2.7 Independence

It may be that under a probability measure P, a certain event A provides no information about another event B: that is, the conditional probability $P(B|A)$ is equal to the prior probability $P(B)$. By the definition of conditional probability, this implies that $P(A \cap B) = P(A)P(B)$. Moving $P(B)$ into the denominator on the left hand side, we then find that $P(A) = P(A|B)$, which is the same as our initial statement but with the roles of A and B reversed. When A and B have these properties, we say that they are *independent*.

Definition 2.12. *Two events A and B are* independent *under a probability distribution P, denoted $A \perp\!\!\!\perp_P B$, if $P(A \cap B) = P(A)P(B)$.*

Thus, independence allows us to *factor* the probability of a conjunction of events, expressing it in terms of the probabilities of individual events. If $P(A) = 0$, then A is independent of all events B, since $P(A \cap B) = P(A)P(B) = 0$. The notion of independence can be extended to random variables as follows.

Definition 2.13. *Let $\mathbf{X}$ and $\mathbf{Y}$ be sets of discrete random variables defined on an outcome space $(\Omega, \mathscr{F})$, and let P be a probability measure on $(\Omega, \mathscr{F})$. Then $\mathbf{X}$ is* independent *of $\mathbf{Y}$ under P, denoted $\mathbf{X} \perp\!\!\!\perp_P \mathbf{Y}$, if $P(\sigma; \tau) = P(\sigma)P(\tau)$ for every finite instantiation σ on $\mathbf{X}$ and every finite instantiation τ on $\mathbf{Y}$.*

It may seem arbitrary to define independence for infinite sets of variables just in terms of finite instantiations, but in fact this criterion can be derived from a more general definition that deals with independence of σ-fields; see Section 1.4 (particularly Theorem 1.4.2) in Durrett [1996]. If we wish to talk about a single random variable X being independent of a set of variables $\mathbf{Y}$, we will simply write $X \perp\!\!\!\perp_P \mathbf{Y}$ in place of the more pedantic $\{X\} \perp\!\!\!\perp_P \mathbf{Y}$. Also, for an event $B \in \mathscr{F}$, we will take $\mathbf{X} \perp\!\!\!\perp_P B$ to mean that $\mathbf{X}$ is independent of the *indicator variable* for B: that is, the variable 1_B such that $1_B(\omega)$ is equal to 1 if $\omega \in B$, and zero otherwise.

Two events may also be *conditionally independent* given another event.

Definition 2.14. *Let P be a probability measure, and let C be an event such that $P(C) > 0$. Two events A and B are* conditionally independent *given C under P, denoted $A \perp\!\!\!\perp_P B \,|\, C$, if $P(A \cap B|C) = P(A|C)P(B|C)$.*

When all the events involved have positive probabilities, saying that A and B are conditionally independent given C is equivalent to saying that $P(A|B, C) = P(A|C)$, or that $P(B|A, C) = P(B|C)$. For discrete random variables, the definition of conditional independence is as follows.

Definition 2.15. *Let* $\mathbf{X}$*,* $\mathbf{Y}$ *and* $\mathbf{Z}$ *be three sets of random variables defined on an outcome space* $(\Omega, \mathscr{F})$ *such that* $\mathbf{Z}$ *is finite, and let* P *be a probability measure on* $(\Omega, \mathscr{F})$*. Then* $\mathbf{X}$ *and* $\mathbf{Y}$ *are* conditionally independent *given* $\mathbf{Z}$ *under* P*, denoted* $\mathbf{X} \perp\!\!\!\perp_P \mathbf{Y} \mid \mathbf{Z}$*, if for every finite instantiation* σ *on* $\mathbf{X}$*, every finite instantiation* τ *on* $\mathbf{Y}$*, and every instantiation* ρ *of* $\mathbf{Z}$ *such that* $P(\rho) > 0$*:*

$$P(\sigma; \tau | \rho) = P(\sigma | \rho) P(\tau | \rho) \tag{2.3}$$

2.3 Graphs and numberings

2.3.1 Directed graphs

A *directed graph* G is a pair (V, E), where V is an arbitrary set of *nodes* or *vertices*, and $E \subseteq V \times V$ is a set of *edges*. Each edge is a pair (u, v) where $u, v \in V$, and is drawn as an arrow from u to v. Note that an edge (u, v) may be a *self-loop* in which $u = v$. The *parents* of a node v, denoted $\mathrm{Pa}(v)$, are those nodes $u \in V$ such that $(u, v) \in E$. A *path* of length n is a sequence of nodes $v_0, v_1, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E$ for each $i < n$. Note that the length of a path is measured by counting edge traversals rather than visits to nodes; a single node v_0 constitutes a path of length zero. An *infinite path* is an infinite sequence of nodes $v_0, v_1, \ldots$ such that $(v_i, v_{i+1}) \in E$ for each $i \in \mathbb{N}$. A *cycle* is a finite path whose first and last element are the same. We will say that a directed graph is *cyclic* if it contains a cycle, and *acyclic* otherwise.

The *ancestors* of a node v, denoted $\mathrm{Anc}(v)$ are those nodes u such that there is a path from u to v. Similarly, the *descendants* of v, denoted $\mathrm{Desc}(v)$, are those nodes u such that there is a path from v to u. The paths used in these definitions may have length zero, so any node v is both an ancestor and a descendant of itself. The

set of *nondescendants* of v is $\text{NonDesc}(v) \triangleq V \setminus \text{Desc}(v)$; this set cannot contain v. We also define the *strict ancestors* of v, denoted $\text{StrictAnc}(v)$, to be the set of nodes u such that there is a non-zero-length path from u to v. The *strict descendants* of v are defined analogously. Thus, the only way v can be a strict ancestor or strict descendant of itself is if it is part of a cycle.

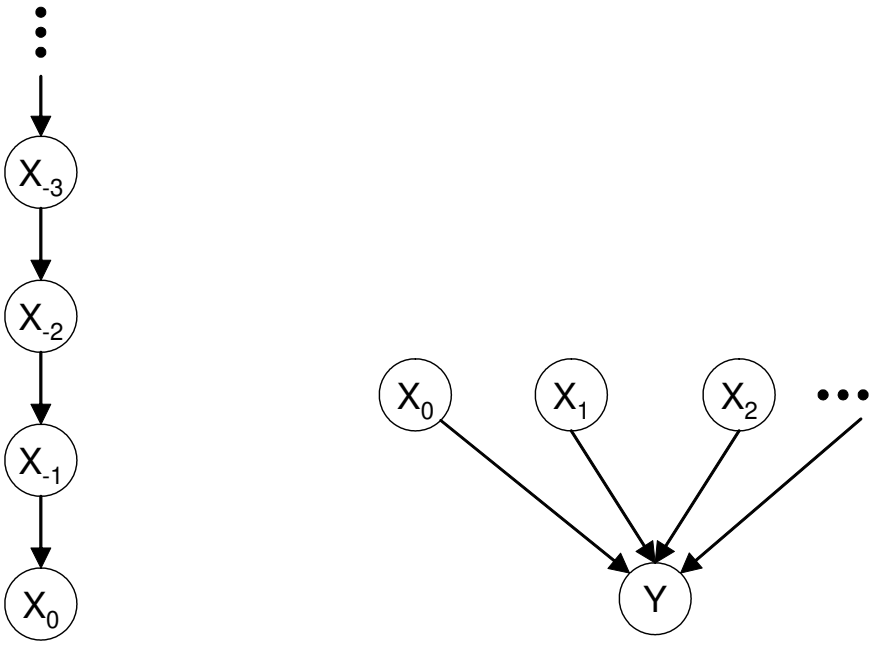
A set of nodes $S \subseteq V$ is an *ancestral set* if it is closed under the parent relation: that is, for each $v \in S$, $\text{Pa}(v) \subseteq S$. Note that a set $\text{NonDesc}(v)$ is always an ancestral set, since if u is a nondescendant of v, then all parents of u are nondescendants of v as well.

2.3.2 Topological numberings

A *numbering* of a set V is a bijection π between V and some prefix of the natural numbers: this will be a finite prefix $\{0, 1, \ldots, n\}$ if V is finite, and the whole set $\mathbb{N}$ if V is infinite. Thus $\pi(v)$ is the number assigned to an element v, and we can list the elements of V as $v_0, v_1, v_2, \ldots$ in order of their numbers. The predecessors of an element v under π are denoted $\text{Pred}_\pi[v]$.

Definition 2.16. *A* topological numbering *of a directed graph $G = (V, E)$ is a numbering π of V such that for each $(u, v) \in E$, $\pi(u) < \pi(v)$.*

It is a standard result that if G is finite and acyclic, then it has a topological numbering. And if G is not countable, then clearly it has no topological numbering. But the case where G is countably infinite is more interesting. Figure 2.1 shows two infinite graphs that do not have topological numberings, even though they are acyclic. Note that these graphs do have topological *orderings*, but certain nodes — all the nodes in graph (a), and node Y in graph (b) — have infinitely many predecessors in all such orderings. A topological numbering could not assign any finite number to these nodes.



(a) (b)

Figure 2.1: Two acyclic graphs that do not have topological numberings.

To ensure that a countably infinite graph has a topological numbering, we have to add another requirement beyond acyclicity: every node must have finitely many ancestors. This fact is a graph-theoretic version of a well-known result in the theory of order relations (see, for example, Exercise 2.15.1 in Fraïssé [2000]). But rather than presenting all the necessary background in order theory, we give a self-contained proof below.

Theorem 2.4. *Let $G = (V, E)$ be a directed graph in which V is countable. There exists a topological numbering of G if and only if G is acyclic and every node in G has finitely many ancestors.*

Proof. Suppose G is acyclic and every node in G has finitely many ancestors. First, let π_0 be an arbitrary numbering of V; this exists since V is countable. Then

construct a topological numbering $v_0, v_1, v_2, \ldots$ of V inductively as follows: for each natural number $n < |V|$, let v_n be the lowest-numbered node in π_0 such that $v_n \notin \{v_0, \ldots, v_{n-1}\}$, but every parent of v_n is in $\{v_0, \ldots, v_{n-1}\}$.

We need to check that such a node v_n exists for each $n < |V|$. Suppose it does not: that is, every node in $V \setminus \{v_0, \ldots, v_{n-1}\}$ has a parent in $V \setminus \{v_0, \ldots, v_{n-1}\}$. Then we can construct an infinite (possibly repeating) sequence of variables $u_1, u_2, \ldots$ in $\mathbf{V} \setminus \{v_0, \ldots, v_{n-1}\}$ such that u_{i+1} is a parent of u_i for each i. If the same node occurs at two different indices i and j in this sequence, then $u_i, \ldots, u_j$ forms a cycle; if the sequence is non-repeating, then u_0 has infinitely many ancestors. Both of these cases are ruled out by our hypotheses. So a node v_n with the desired properties must exist.

We also need to check that this numbering $v_0, v_1, \ldots$ includes every node in V. By hypothesis, every node has finitely many ancestors, so we can proceed by induction on the size of a node's strict ancestor set. If a node w has zero strict ancestors, then at any step in the construction where $w \notin \{v_0, \ldots, v_{n-1}\}$, w is eligible to be chosen as v_n. Nodes are chosen according to the numbering π_0, so at most $\pi_0(w)$ nodes can be chosen before w. Thus, w must eventually be chosen. Now assume all nodes w with $|\text{StrictAnc}(w)| \leq m$ are included in $v_0, v_1, \ldots$, and consider a variable w with $m + 1$ strict ancestors. If u is a strict ancestor of w, then StrictAnc(u) must be a proper subset of StrictAnc(w): in particular, u cannot be in StrictAnc(u), because the graph is acyclic. So by the inductive hypothesis, u is included in $v_0, v_1, \ldots$. Since this holds for each $u \in \text{StrictAnc}(w)$ and StrictAnc(w) is finite, it follows that there is some n such that $\{v_0, \ldots, v_{n-1}\}$ includes all of StrictAnc(w). So w must be included in the sequence $v_0, v_1, \ldots$ with an index less than or equal to $n + \pi_0(w)$.

For the converse, suppose π is a topological numbering of G. Assume for contradiction that G contains a cycle $v_1, v_2, \ldots, v_n$ with $v_1 = v_n$. Then by the definition of a topological ordering, $\pi(v_i) < \pi(v_{i+1})$ for all $i < n$. But this implies $\pi(v_1) < \pi(v_1)$, which is impossible. Now suppose G contains a node v with infinitely many ances-

tors. The fact that π is topological implies $\pi(u) < \pi(v)$ for each strict ancestor u of v. But then there are infinitely many elements of V with numbers less than $\pi(v)$. Given that π is a bijection, this is also impossible. □

2.3.3 König's infinity lemma

Theorem 2.4 makes use of the condition that every node in a graph G has finitely many ancestors. However, it is not always obvious whether a specified graph satisfies this condition or not. In this section, we provide an equivalent condition that may be simpler to check. An *infinite receding chain* in a directed graph $G = (V, E)$ is an infinite sequence $v_0, v_1, \ldots$ of distinct elements of V, such that v_{i+1} is a parent of v_i for each i. For example, the entire graph in Figure 2.1(a) is an infinite receding chain. The main result of this section is that in a graph G, every node has finitely many ancestors if and only if G contains no infinite receding chains and every node has finitely many parents.

This fact is a straightforward consequence of König's infinity lemma [König, 1926]. The simplest statement of this lemma says that if G is an infinite, connected, *undirected* graph in which every node has finitely many neighbors, then G contains an infinite path [König, 1927; Franchella, 1997]. For our purposes, however, it is more convenient to use a version of the lemma that deals with a directed relation. The following statement is adapted from Nash-Williams [1967].

Theorem 2.5 (König's Lemma). *Let $S_0, S_1, \ldots$ be an infinite sequence of disjoint, non-empty, finite sets of nodes in a directed graph $G = (V, E)$. Suppose that for each natural number i, each node in S_{i+1} is a parent of some node in S_i. Then G contains an infinite receding chain $v_0, v_1, \ldots$ where $v_i \in S_i$ for each i.*

König's lemma allows us to prove the following proposition.

Proposition 2.6. *In a directed graph G, every node has finitely many ancestors if and only if G contains no infinite receding chains and every node in G has finitely many parents.*

Proof. The "only if" direction is trivial: if every node in G has finitely many ancestors, then clearly every node has finitely many parents, and there can be no infinite chain because the nodes in such a chain would have infinitely many ancestors. For the "if" direction, suppose G contains no infinite receding chain and every node in G has finitely many parents. Then consider any node v; we will show that v has finitely many ancestors.

To fit the conditions of König's lemma, define a sequence of sets $S_0, S_1, \ldots$ where S_i is the set of nodes u such that there is a path from u to v of length i and not of any shorter length. By definition, for every $u \in \mathrm{Anc}(v)$, there is a path of some length from u to v. So $\{S_0, S_1, \ldots\}$ is a partition of $\mathrm{Anc}(v)$. Note that every node in S_{i+1} is a parent of some node in S_i, since a node with a length-$(i+1)$ path to v must have a child with a length-i path to v. Also, the set S_0 is necessarily finite because it consists only of v itself. And if S_i is finite then so is S_{i+1}, because each node in S_i has only finitely many parents. So by induction, all the sets $S_0, S_1, \ldots$ are finite. Also, we know that G contains no infinite receding chain. Thus, by König's lemma, the only possibility is that S_i is empty for some i. This implies that S_j is empty for all $j > i$, and thus $\mathrm{Anc}(v)$ can be partitioned into a finite sequence of finite sets $S_0, \ldots, S_{i-1}$. So $\mathrm{Anc}(v)$ is finite. □

2.4 Bayesian networks

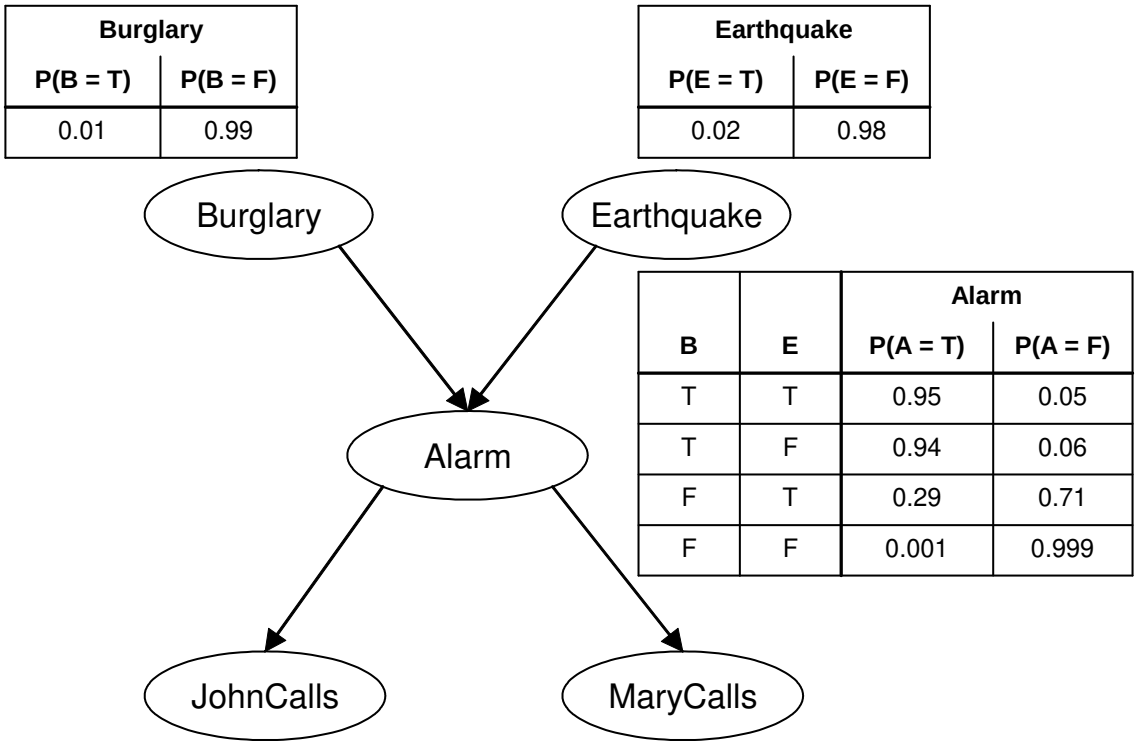
2.4.1 Introduction

If we want to specify a joint distribution for a set of random variables, then in principle we can just list all the complete instantiations and specify a probability for each one (assuming the number of variables is finite and each has a finite range). As long as the probabilities we specify add up to one, then this specification will indeed define a unique distribution. However, this approach is not feasible for real-world scenarios: the number of probabilities one must specify grows exponentially with the number of variables. This is a problem both computationally, because the model will take exponential space and require exponential time to answer queries, and statistically, because the number of probabilities to estimate from data will be exponentially large.

Bayesian networks (BNs) [Pearl, 1988; Cowell *et al.*, 1999] (also called *belief networks* or *directed graphical models*) allow a modeler to avoid this exponential blow-up by exploiting independence properties. We will introduce BNs with the following simple example, first used by Pearl [1988].

Example 2.1. *Suppose I have a home alarm system that is designed to be triggered by would-be burglars, but can also be set off by small earthquakes, which are common where I live. If my alarm goes off while I am at work, my neighbors John and Mary may call to let me know. My beliefs about this scenario can be be formalized with a probability distribution over the product space of five variables:* Burglary, Earthquake, Alarm, JohnCalls, *and* MaryCalls. *Each of these variables is Boolean, taking values in the set* $\{\mathsf{T}, \mathsf{F}\}$.

Figure 2.2 shows a BN for this example. A BN consists of two parts, the *BN structure* and the *conditional probability distributions* (CPDs). The BN structure



Burglary	
P(B = T)	P(B = F)
0.01	0.99

Earthquake	
P(E = T)	P(E = F)
0.02	0.98

		Alarm	
B	E	P(A = T)	P(A = F)
T	T	0.95	0.05
T	F	0.94	0.06
F	T	0.29	0.71
F	F	0.001	0.999

	JohnCalls	
A	P(J = T)	P(J = F)
T	0.9	0.1
F	0.05	0.95

	MaryCalls	
A	P(M = T)	P(M = F)
T	0.7	0.3
F	0.01	0.99

Figure 2.2: A Bayesian network for Example 2.1, including the BN structure and conditional probability tables.

is a directed graph with a node for each random variable. The edges in the graph represent probabilistic dependencies, in a sense that we will make precise below. We will write $\mathrm{Pa}_{\mathcal{B}}(X)$ to denote the parents of a variable X in a BN $\mathcal{B}$.

For each variable X, $\mathcal{B}$ specifies a conditional probability distribution (see Section 2.2.6) for X given $\mathrm{Pa}_{\mathcal{B}}(X)$. When X and all its parents have finite ranges, a CPD for X can be represented as a *conditional probability table* (CPT) with a row for each instantiation of $\mathrm{Pa}_{\mathcal{B}}(X)$. This is illustrated in Figure 2.2. Note that in this example, the CPTs contain only 20 probability values. In fact, since the values in each row of each CPT must sum to one, this representation has only 10 free parameters. By contrast, a table listing probabilities for all 32 instantiations of these five binary variables would have 31 free parameters. Thus, even for this small example, the BN is considerably more compact than a naive representation. The advantage of a BN increases with the number of variables: while an explicit representation of a joint distribution for n k-ary variables has $k^n - 1$ parameters, a BN representation in which each variable has at most m parents has only $O(nk^m)$ parameters.

2.4.2 Syntax and semantics for infinite BNs

We now give a more formal treatment of BN syntax and semantics. We limit ourselves to discrete random variables, but we allow the number of variables to be infinite. We require that each variable have finitely many parents: this is because we wish to continue working with the elementary definition of conditional probability (as given in Section 2.2.6) rather than a more advanced definition based on σ-fields. Although the main results about BNs are well known, we give proofs here for two reasons. First, the standard references [Pearl, 1988; Cowell *et al.*, 1999] do not deal with infinite sets of variables. Previous treatments of infinite BNs [Jaeger, 1998; Pfeffer, 2000; Kersting and De Raedt, 2001a; Laskey, 2006] have been tied

to particular representation languages, making it awkward to cite their theorems in other contexts. Second, our development of partition-based models in Chapter 3 will be analogous to the sequence of results here. Understanding the line of reasoning in this section should make Chapter 3 easier to understand.

Definition 2.17. *A* Bayesian network structure *over a set of variables $\mathcal{V}$ is a directed graph in which the nodes are the elements of $\mathcal{V}$, and each node has finitely many parents.*

Definition 2.18. *Let $\mathcal{V}$ be a countable set of discrete random variables that are the coordinate variables of a product space $(\Omega, \mathscr{F})$. A* Bayesian network *(BN) $\mathcal{B}$ for $\mathcal{V}$ consists of:*

- *a Bayesian network structure $\mathcal{G}_\mathcal{B}$ over $\mathcal{V}$;*
- *for each $X \in \mathcal{V}$, a conditional probability distribution $c_\mathcal{B}^X$ for X given $\mathrm{Pa}_\mathcal{B}(X)$.*

This definition assumes that the variables in $\mathcal{V}$ are all defined on a common outcome space $(\Omega, \mathscr{F})$, and furthermore that $(\Omega, \mathscr{F})$ is a product space with $\mathcal{V}$ as its coordinate variables (see Section 2.2.1). A BN $\mathcal{B}$ for $\mathcal{V}$ is a declarative representation of a probability measure P on $(\Omega, \mathscr{F})$. One assertion $\mathcal{B}$ makes about P is that for each variable X, the CPD $c_\mathcal{B}^X$ is a version of $P(X|\mathrm{Pa}_\mathcal{B}(X))$. However, specifying a version of $P(X|\mathrm{Pa}_\mathcal{B}(X))$ for each $X \in \mathcal{V}$ typically does not yield a complete specification of P. This is apparent in a very simple example: suppose $\mathcal{B}$ has just two variables, X and Y, with no edges between them. Then specifying the CPDs for X and Y given their parents just amounts to specifying the marginal distributions for X and Y, and there can be many different probability measures that yield the same marginals for these two variables. Thus, we define the semantics of a BN not just in terms of CPDs, but also in terms of independence properties.

Definition 2.19. *Let $\mathcal{B}$ be a BN over a set of random variables $\mathcal{V}$ defined on $(\Omega, \mathscr{F})$. A probability measure P on $(\Omega, \mathscr{F})$* satisfies *$\mathcal{B}$ if, for each $X \in \mathcal{V}$,*

i. $c_{\mathcal{B}}^X$ is a version of $P(X|\mathrm{Pa}_{\mathcal{B}}(X))$; and

ii. $X \perp\!\!\!\perp_P \mathrm{NonDesc}_{\mathcal{B}}(X) \,|\, \mathrm{Pa}_{\mathcal{B}}(X)$.

Thus, in addition to specifying CPDs, a BN asserts the *directed local Markov property* [Howard and Matheson, 1984; Kiiveri *et al.*, 1984]: each variable is independent of its nondescendants given its parents. A BN is *well-defined* if there is exactly one probability measure that satisfies it. The main result of this section is that if a BN has a topological numbering — or equivalently (by Theorem 2.4), if it is acyclic and each of its nodes has finitely many ancestors — then it is well-defined. Thus these BNs perform the desired task of fully defining a probability measure over the outcome space.

We begin by showing that the local Markov property implies a certain factorization property for instantiations whose variables form a finite prefix of a topological numbering. If $\mathcal{B}$ has finitely many variables and has a topological numbering, then this factorization applies to complete instantations of $\mathcal{B}$'s variables. In fact, this factorization property for complete instantiations is often used as the definition of BN semantics [Lauritzen *et al.*, 1990].

Lemma 2.7. *Suppose $X_0, X_1, \ldots$ is a topological numbering of a BN $\mathcal{B}$. Let P be a probability measure that satisfies $\mathcal{B}$. Then for any instantiation σ such that* $\mathrm{vars}\,(\sigma)$ *is a finite prefix of $X_0, X_1, \ldots$,*

$$P(\sigma) = \prod_{X \,\in\, \mathrm{vars}(\sigma)} c_{\mathcal{B}}^X(\sigma_X, \sigma[\mathrm{Pa}_{\mathcal{B}}(X)]) \tag{2.4}$$

Proof. We proceed by induction on $|\mathrm{vars}\,(\sigma)\,|$. For $|\mathrm{vars}\,(\sigma)\,| = 0$, we just have $P(\top) = 1$, which is true by the definition of a probability measure. Now sup-

pose Equation 2.4 holds for all instantiations of $\{X_0, \ldots, X_{n-1}\}$, and consider an instantiation σ of $\{X_0, \ldots, X_n\}$. If $P(\sigma_{-X_n}) > 0$, then the definition of conditional probability tells us that:

$$P(\sigma) = P(X_n = \sigma_{X_n} \mid \sigma_{-X_n})\, P(\sigma_{-X_n})$$

Because $X_0, X_1, \ldots$ is a topological numbering of $\mathcal{B}$, we know $\mathrm{Pa}_{\mathcal{B}}(X_n) \subseteq \{X_0, \ldots, X_{n-1}\}$. We also know $\{X_0, \ldots, X_{n-1}\} \subseteq \mathrm{NonDesc}_{\mathcal{B}}(X_n)$, so by condition (ii) of Definition 2.19,

$$P(\sigma) = P(X_n = \sigma_{X_n} \mid \sigma[\mathrm{Pa}_{\mathcal{B}}(X_n)])\, P(\sigma_{-X_n})$$

By condition (i) of Definition 2.19, $c_{\mathcal{B}}^{X_n}$ is a version of $P(X_n \mid \mathrm{Pa}_{\mathcal{B}}(X_n))$. So:

$$P(\sigma) = c_{\mathcal{B}}^{X_n}(\sigma_{X_n}, \sigma[\mathrm{Pa}_{\mathcal{B}}(X_n)])\, P(\sigma_{-X_n})$$

This equation also holds when $P(\sigma_{-X_n}) = 0$, because then both sides of the equation are zero. Now by the inductive hypothesis,

$$P(\sigma) = c_{\mathcal{B}}^{X_n}(\sigma_{X_n}, \sigma[\mathrm{Pa}_{\mathcal{B}}(X_n)]) \prod_{X \in \{X_0, \ldots, X_{n-1}\}} c_{\mathcal{B}}^{X}(\sigma_X, \sigma[\mathrm{Pa}_{\mathcal{B}}(X)])$$

Moving the $c_{\mathcal{B}}^{X_n}$ factor into the product expression yields Equation 2.4, as desired. □

Thus, if $X_0, X_1, \ldots$ is a topological numbering for a BN $\mathcal{B}$, then $\mathcal{B}$ uniquely determines the probabilities of all instantiations of finite prefixes of $X_0, X_1, \ldots$. Using Kolmogorov's extension theorem (Theorem 2.2), we will be able to show that there is a unique probability measure on $\mathcal{B}$'s outcome space that assigns the specified probabilities to these instantiations. However, we must also show that assigning the right probabilities to these instantiations is a sufficient condition for satisfying $\mathcal{B}$. We do this in two steps. First, we show that if a probability measure P satisfies

Equation 2.4 for all instantiations of prefixes of $X_0, X_1, \ldots$, then in fact it satisfies Equation 2.4 for all instantiations of finite, ancestral sets. This is the broadest class of instantiations σ for which Equation 2.4 even makes sense: $\text{vars}(\sigma)$ must be ancestral so that $\sigma[\text{Pa}_{\mathcal{B}}(X)]$ is defined for each $X \in \text{vars}(\sigma)$, and $\text{vars}(\sigma)$ must be finite so that we do not get an infinite product expression, which would yield zero for most instantiations. Second, we show that if P satisfies Equation 2.4 for all instantiations of finite, ancestral sets, then it has the CPDs and independence properties necessary to satisfy $\mathcal{B}$.

Lemma 2.8. *Suppose $X_0, X_1, \ldots$ is a topological numbering of a BN $\mathcal{B}$. Let P be a probability measure on $\mathcal{B}$'s outcome space such that $P(\sigma)$ satisfies Equation 2.4 whenever $\text{vars}(\sigma)$ is a finite prefix of $X_0, X_1, \ldots$. Then $P(\sigma)$ satisfies Equation 2.4 for every instantiation σ such that $\text{vars}(\sigma)$ is a finite, ancestral set in $\mathcal{B}$.*

Proof. Consider any instantiation σ such that $\text{vars}(\sigma)$ is a finite, ancestral set in $\mathcal{B}$. We proceed by induction on the index of the highest-numbered element of $\text{vars}(\sigma)$ in $X_0, X_1, \ldots$. The conclusion is trivially true for $\sigma = \top$; our base case is where the highest-numbered element of $\text{vars}(\sigma)$ is X_0. In this case, $\text{vars}(\sigma)$ is a finite prefix of $X_0, X_1, \ldots$, so the conclusion is immediate.

Now suppose the conclusion holds for instantiations whose highest-numbered variable has index at most $n-1$. Let σ be an instantiation of a finite, ancestral set whose highest-numbered element is X_n. Now let R be the set of instantiations of $\{X_0, \ldots, X_{n-1}\}$ that do not contradict σ. Because $\{X_0, \ldots, X_{n-1}\}$ and $\{X_0, \ldots, X_n\}$ are both finite prefixes of $X_0, X_1, \ldots$, we can apply Equation 2.4 to conclude that for each $\rho \in R$:

$$P(\rho; X_n = \sigma_{X_n}) = P(\rho)\, c_{\mathcal{B}}^{X_n}(\sigma_{X_n}, \rho[\text{Pa}_{\mathcal{B}}(X_n)])$$

Since $\text{vars}(\sigma)$ is an ancestral set, we know $\text{Pa}_{\mathcal{B}}(X_n) \subseteq \text{vars}(\sigma)$, so $\rho[\text{Pa}_{\mathcal{B}}(X_n)] = \sigma[\text{Pa}_{\mathcal{B}}(X_n)]$ for all $\rho \in R$. Thus we can rewrite the $c_{\mathcal{B}}^{X_n}$ factor in our previous equation

without using ρ:

$$P(\rho; X_n = \sigma_{X_n}) = P(\rho)\, c_{\mathcal{B}}^{X_n}(\sigma_{X_n}, \sigma[\mathrm{Pa}_{\mathcal{B}}(X_n)])$$

Our choice of R ensures that $\{\mathrm{ev}\,(\rho) \;:\; \rho \in R\}$ is a partition of $\mathrm{ev}\,(\sigma_{-X_n})$, and $\{\mathrm{ev}\,(\rho; X_n = \sigma_{X_n}) \;:\; \rho \in R\}$ is a partition of $\mathrm{ev}\,(\sigma)$. So summing over $\rho \in R$ on both sides of our previous equation yields:

$$P(\sigma) = P(\sigma_{-Y}) c_{\mathcal{B}}^{X_n}(\sigma_{X_n}, \sigma[\mathrm{Pa}_{\mathcal{B}}(X_n)])$$

Then by the inductive hypothesis,

$$P(\sigma) = c_{\mathcal{B}}^{X_n}(\sigma_{X_n}, \sigma[\mathrm{Pa}_{\mathcal{B}}(X_n)]) \prod_{X \in \mathrm{vars}(\sigma_{-X_n})} c_{\mathcal{B}}^{X}(\sigma_X, \sigma[\mathrm{Pa}_{\mathcal{B}}(X)])$$

Moving the $c_{\mathcal{B}}^{X_n}$ factor into the product expression yields Equation 2.4. □

Lemma 2.9. *Let $\mathcal{B}$ be a BN that has a topological numbering, and let P be a probability measure on $\mathcal{B}$'s outcome space. If $P(\sigma)$ satisfies Equation 2.4 for each instantiation σ such that* $\mathrm{vars}\,(\sigma)$ *is a finite, ancestral set in $\mathcal{B}$, then P satisfies $\mathcal{B}$.*

Proof. Consider any variable X in $\mathcal{B}$. Let x be any value in $\mathrm{range}\,(X)$, ρ be any instantiation of $\mathrm{Pa}_{\mathcal{B}}(X)$ such that $P(\rho) > 0$, and σ be any finite instantiation on $\mathrm{NonDesc}_{\mathcal{B}}(X)$. We will show that $P(\sigma; X = x \,|\, \rho) = c_{\mathcal{B}}^{X}(x, \rho) P(\sigma|\rho)$. Let $\mathbf{A} = \bigcup\{\mathrm{Anc}_{\mathcal{B}}(W) \;:\; W \in (\mathrm{vars}\,(\sigma) \cup \mathrm{Pa}_{\mathcal{B}}(X))\}$. Let T be the set of instantiations τ of $\mathbf{A}$ such that $\tau[\mathrm{Pa}_{\mathcal{B}}(X)] = \rho$ and $\tau[\mathrm{vars}\,(\sigma)] = \sigma$ (this is an empty set if σ contradicts ρ). Thus:

$$P(\rho; \sigma; X = x) = \sum_{\tau \in T} P(\tau; X = x)$$

Note that $\mathbf{A}$ is an ancestral set. Also, since $\mathcal{B}$ has a topological numbering, we know each node has finitely many ancestors, so $\mathbf{A}$ is finite. And $\mathrm{Pa}_{\mathcal{B}}(X) \subseteq \mathbf{A}$, so both $\mathbf{A}$ and $\mathbf{A} \cup \{X\}$ are finite, ancestral sets. Thus we can apply Equation 2.4 to both τ and $(\tau; X = x)$ for each $\tau \in T$. We can also be sure that X is not in $\mathbf{A}$: X is not an ancestor of any variable in $\mathrm{vars}(\sigma)$ because $\mathrm{vars}(\sigma) \subseteq \mathrm{NonDesc}_{\mathcal{B}}(X)$, and X is not an ancestor of any variable in $\mathrm{Pa}_{\mathcal{B}}(X)$ because $\mathcal{B}$ has a topological numbering and hence is acyclic. So applying Equation 2.4 allows us to conclude that $P(\tau; X = x) = c^X_{\mathcal{B}}(x, \rho)P(\tau)$. Therefore:

$$P(\rho; \sigma; X = x) = c^X_{\mathcal{B}}(x, \rho) \sum_{\tau \in T} P(\tau)$$

Now by our choice of T:

$$\begin{aligned} P(\rho; \sigma; X = x) &= c^X_{\mathcal{B}}(x, \rho)\, P(\rho; \sigma) \\ P(\sigma; X = x \mid \rho) &= c^X_{\mathcal{B}}(x, \rho)\, P(\sigma|\rho) \end{aligned}$$

This is the equation we set out to prove. Taking $\sigma = \top$ yields $P(X = x \mid \rho) = c^X_{\mathcal{B}}(x, \rho)$. Since this holds for all $x \in \mathrm{range}(X)$ and $\rho \in \mathrm{range}(\mathrm{Pa}_{\mathcal{B}}(X))$ such that $P(\rho) > 0$, we can conclude that $c^X_{\mathcal{B}}$ is a version of $P(X|\mathrm{Pa}_{\mathcal{B}}(X))$. Applying this conclusion to our last equation gives:

$$P(\sigma; X = x \mid \rho) = P(x|\rho)P(\sigma|\rho)$$

This holds for any finite instantiation σ on $\mathrm{NonDesc}_{\mathcal{B}}(X)$, so we have the independence property $X \perp\!\!\!\perp_P \mathrm{NonDesc}_{\mathcal{B}}(X) \mid \mathrm{Pa}_{\mathcal{B}}(X)$. Thus, P satisfies $\mathcal{B}$. □

Given these lemmas, we now have two conditions on a probability measure P that are equivalent to the statement that P satisfies a given BN.

Theorem 2.10. *Let $\mathcal{B}$ be a BN that has a topological numbering, and let $X_0, X_1, \ldots$ be one such numbering. Let P be a probability measure on $\mathcal{B}$'s outcome space. Then the following three conditions are equivalent:*

1. *P satisfies $\mathcal{B}$ (in the sense of Definition 2.19).*

2. *for every instantiation σ such that $\text{vars}(\sigma)$ is a finite prefix of $X_0, X_1, \ldots$, $P(\sigma)$ satisfies Equation 2.4.*

3. *for every instantiation σ such that $\text{vars}(\sigma)$ is a finite, ancestral set in $\mathcal{B}$, $P(\sigma)$ satisfies Equation 2.4.*

Proof. By Lemma 2.7, (1) implies (2). By Lemma 2.8, (2) implies (3). And by Lemma 2.9), (3) implies (1). □

We can now use Kolmogorov's extension theorem to show that for a BN with a topological ordering, exactly one probability measure satisfies these three equivalent conditions. This result is given in Section 5.4 of Pfeffer's thesis [2000] under the assumption that each variable has a finite range, which removes the need to appeal to Kolmogorov's theorem. Kersting and De Raedt [2001b] cover the general case.

Theorem 2.11. *If a BN has a topological numbering, then it is well-defined.*

Proof. Let $\mathcal{B}$ be a BN that has a topological numbering, and let $X_0, X_1, \ldots$ be one such numbering. Let $\mathcal{V}$ be the set of variables in $\mathcal{B}$ and $(\Omega, \mathscr{F})$ be $\mathcal{B}$'s outcome space. We must show that there is a unique probability measure on $(\Omega, \mathscr{F})$ that satisfies $\mathcal{B}$. By Definition 2.18, $(\Omega, \mathscr{F})$ is a product space with $\mathcal{V}$ as its coordinate variables, so $X_0, X_1, \ldots$ can serve as the numbering of the coordinate variables required by Kolmogorov's extension theorem (Theorem 2.2). For each natural number $n \leq |\mathcal{V}|$,

define a function f_n on instantiations of $\{X_0, \ldots, X_{n-1}\}$ as follows:

$$f_n(\sigma) \triangleq \prod_{X \in \text{vars}(\sigma)} c_{\mathcal{B}}^{X}(\sigma_X, \sigma[\text{Pa}_{\mathcal{B}}(X)])$$

We must show that these functions f_n satisfy the conditions of Theorem 2.2. Since each $f_n(\sigma)$ is a product of values returned by CPDs, it is clear that f_n returns values in $[0, 1]$. Also, $f_0(\top)$ is an empty product, which equals 1. Now consider any natural number $n < |\mathcal{V}|$ and any instantiation σ of $\{X_0, \ldots, X_{n-1}\}$.

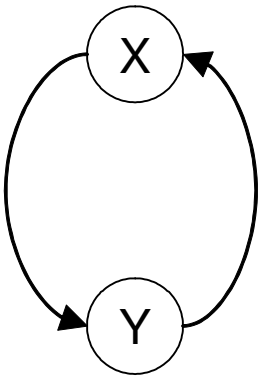
$$\begin{aligned}
\sum_{x_n \in \text{range}(X_n)} f_{n+1}(\sigma; X_n = x_n) &= \sum_{x_n \in \text{range}(X_n)} f_n(\sigma) c_{\mathcal{B}}^{X_n}(x_n, \sigma[\text{Pa}_{\mathcal{B}}(X_n)]) \\
&= f_n(\sigma) \sum_{x_n \in \text{range}(X_n)} c_{\mathcal{B}}^{X_n}(x_n, \sigma[\text{Pa}_{\mathcal{B}}(X_n)]) \\
&= f_n(\sigma)
\end{aligned}$$

The last step here follows from the definition of a CPD (Definition 2.10). So the functions f_n satisfy all the conditions of Theorem 2.2, which implies there is exactly one probability measure P on $(\Omega, \mathscr{F})$ such that $P(\sigma) = f_n(\sigma)$ for all $n \leq |\mathcal{V}|$ and all $\sigma \in \text{range}(\{X_0, \ldots, X_{n-1}\})$. By the definition of f_n, $P(\sigma) = f_n(\sigma)$ just when $P(\sigma)$ satisfies Equation 2.4. So by Theorem 2.10 (specifically the fact that condition (2) implies condition (1)), there is exactly one probability measure that satisfies $\mathcal{B}$. □

By Theorem 2.4, we get the following corollary.

Corollary 2.12. *If a BN is acyclic and every one of its variables has finitely many ancestors, then it is well-defined.*

Thus, we have identified a class of BNs that are guaranteed to be well-defined.



Y	X	
	P(X = T)	P(X = F)
T	0.9	0.1
F	0.1	0.9

X	Y	
	P(Y = T)	P(Y = F)
T	0.1	0.9
F	0.9	0.1

Figure 2.3: A cyclic BN that is not satisfied by any probability distribution.

2.4.3 Ill-defined BNs

Our well-definedness results or BNs only apply when the network is acyclic and each of its nodes has finitely many ancestors. It is instructive to see what can go wrong when these conditions are violated. First, let $\mathcal{B}$ be the cyclic BN in Figure 2.3. It turns out that there is no probability measure P such that $c^X_{\mathcal{B}}$ and $c^Y_{\mathcal{B}}$ are versions of $P(X|Y)$ and $P(Y|X)$, respectively. To be consistent with the first row of X's CPD, P would need to satisfy $P(X=T,Y=T) = \frac{0.9}{0.1} \times P(X=F,Y=T) = 9 \times P(X=F,Y=T)$. And to be consistent with the second row of Y's CPD, it would need to have $P(X=F,Y=T) = 9 \times P(X=F,Y=F)$. So we have $P(X=T,Y=T) = 81 \times P(X=F,Y=F)$. However, the same argument with the second row of X's CPD and the first row of Y's CPD gives us the opposite conclusion: $P(X=F,Y=F) = 81 \times P(X=T,Y=T)$. These constraints can be satisfied only if $P(X=T,Y=T)$ and $P(X=F,Y=F)$ are both zero — but then the CPDs imply that $P(X=T,Y=F)$

and $P(X = F, Y = T)$ are both zero as well, which is impossible since a probability measure must sum to one.

As another example, consider a BN $\mathcal{B}$ that contains an infinite receding chain $X_0, X_{-1}, X_{-2}, \ldots$, as shown in Figure 2.1(a). Suppose all the variables are Boolean and the CPDs are deterministic: specifically, let $c_{\mathcal{B}}^{X_i}(T, (X_{i-1} = T)) = 1$ and let $c_{\mathcal{B}}^{X_i}(F, (X_{i-1} = F)) = 1$. Then there are *two* probability measures that satisfy $\mathcal{B}$: one where all the variables are true with probability one, and one where they are all false with probability one. In general, an infinite receding chain in a BN can be thought of as a Markov chain with no specified distribution for its initial state. As pointed out by Pfeffer and Koller [2000], the BN is satisfied by multiple probability measures whenever this Markov chain has multiple stationary distributions.

2.4.4 Extensions

Most treatments of Bayesian networks allow continuous random variables [Cowell *et al.*, 1999]. It is fairly straightforward to extend the development we have given here to handle continuous variables by letting the CPDs define conditional densities, and letting $c_{\mathcal{B}}$ define a probability density for each finite, ancestral set of variables. We have refrained from using density functions here because we will not be using them in Chapter 3, where the extension from discrete to continuous variables is more complicated.

It is also possible to relax the requirement that every variable have finitely many parents (even without representing the conditions under which dependencies are active, as we do in Chapter 3). If we allowed infinite parent sets, we would have to introduce more measure-theoretic machinery to define what it means for $c_{\mathcal{B}}^{X}$ to be a version of $P(X|\mathrm{Pa}_{\mathcal{B}}(X))$, since individual instantiations of $\mathrm{Pa}_{\mathcal{B}}(X)$ will typically have probability zero. We would also have to give up topological numberings and instead

use *well-founded orderings* [Ciesielski, 1997], in which cycles and infinite receding chains are still disallowed but a variable may have infinitely many predecessors. Jaeger [1998] outlines this approach, but only gives details for a case where the variables can be divided into finitely many levels, such that the ancestors of each variable are in preceding levels. Laskey [2006] gives a well-definedness proof for the case where for each variable X, there is an upper bound on the length of paths ending at X. This restriction is slightly stronger than the requirement of a well-founded topological ordering, which allows a variable to have incoming paths of all finite lengths.

2.5 Sampling methods for probabilistic inference

Probabilistic inference is the task of computing the posterior probability of an event given some evidence. Specifically, suppose we are given a probability measure P on an outcome space $(\Omega, \mathscr{F})$, an *evidence event* $E \in \mathscr{F}$ with $P(E) > 0$, and a *query event* $Q \in \mathscr{F}$. For instance, in the Bayesian network of Figure 2.2, the evidence might be represented by the instantiation ($\mathsf{JohnCalls} = \mathsf{T}$, $\mathsf{MaryCalls} = \mathsf{F}$), and the query might be ($\mathsf{Burglary} = \mathsf{T}$). The inference task, then, is to compute $P(Q|E)$.

We restrict ourselves to discrete outcome spaces throughout this section. Given this assumption, $P(Q|E)$ can be written as:

$$P(Q|E) = \frac{P(Q \cap E)}{P(E)} = \frac{\sum_{\omega \in Q \cap E} P(\omega)}{\sum_{\omega \in E} P(\omega)}$$

In principle, if the set E is finite, we can enumerate all the outcomes $\omega \in E$ and compute these sums directly. But in cases of practical interest, the number of outcomes consistent with the evidence can be extremely large: in a Bayesian network, it is exponential in the number of unobserved variables. The conditional independence prop-

erties expressed by a BN mitigate this problem to some extent. It is possible to use dynamic programming algorithms such as *variable elimination* [Shachter *et al.*, 1990; Zhang and Poole, 1994] or the *junction tree* method [Lauritzen and Spiegelhalter, 1988] to do inference on a BN without enumerating each outcome separately. However, the complexity of these methods is still exponential in the *treewidth* of the network, which grows as the graph becomes more highly connected [Dechter, 1999].

Thus, there has been a great deal of interest in approximation algorithms for probabilistic inference. The ones we discuss in this thesis are *Monte Carlo* algorithms, based on random sampling (or technically, pseudorandom number generation on a computer). One popular alternative to Monte Carlo methods is *loopy belief propagation* [Murphy *et al.*, 1999], but it fails to converge on certain models. *Variational methods* [Jordan *et al.*, 1999] are another family of deterministic approximation algorithms. This family includes general-purpose methods for BNs with a certain class of CPDs [Xing *et al.*, 2003]; for other models, one must attempt to derive equations by hand.

The general-purpose Monte Carlo methods we discuss below can be applied to a very wide range of models, and they are guaranteed to converge to correct probabilities if one generates enough samples. They do have drawbacks: convergence can be slow, and it can be difficult to tell whether convergence has occurred. Still, sampling-based methods are the best algorithms available for many probabilistic inference tasks.

2.5.1 Rejection sampling

For any evidence event E with $P(E) > 0$, we can consider the posterior probability distribution P_E on E such that $P_E(\omega) = P(\omega)/P(E)$. Then $P(Q|E)$ is simply $P_E(Q \cap E)$. If we can generate samples from P_E directly, then we can apply a very

simple algorithm to approximate $P_E(Q \cap E)$: generate a sequence of independent samples $s_1, s_2, \ldots, s_N$ from P_E, and estimate $P_E(Q \cap E)$ by the fraction of samples that are in Q. If we write $\mathbf{1}_S$ for the *indicator variable* that returns 1 on outcomes $\omega \in S$ and 0 on other outcomes, then we can write this estimate as:

$$\hat{p}_N = \frac{1}{N} \sum_{n=1}^{N} \mathbf{1}_{Q \cap E}(s_n) \tag{2.5}$$

The strong law of large numbers (see, *e.g.*, Billingsley [1995]) guarantees that with probability one, $\hat{p}_N$ converges to $P_E(Q \cap E)$ (which is the expectation of $\mathbf{1}_{Q \cap E}$) as $N \to \infty$.

In many cases, it is not easy to generate samples directly from P_E. In a BN, for example, the only case where such sampling is straightforward is when the evidence is just an instantiation of some root nodes. However, we can often generate samples from the full outcome space Ω according to a *proposal distribution* q such that $P_E(\omega)$ is proportional to $q(\omega)$ for $\omega \in E$. For example, q might be the prior distribution P. Then we can approximate $P_E(Q \cap E)$ using the *rejection sampling* method [Rubinstein, 1981]. The idea is to reject samples that fall outside of E, and compute our estimate based only on samples in E. Specifically, the estimate of $P_E(Q \cap E)$ after N samples is:

$$\hat{p}_N = \begin{cases} 0 & \text{if } \sum_{n=1}^{N} \mathbf{1}_E(s_n) = 0 \\ \dfrac{\sum_{n=1}^{N} \mathbf{1}_{Q \cap E}(s_n)}{\sum_{n=1}^{N} \mathbf{1}_E(s_n)} & \text{otherwise} \end{cases} \tag{2.6}$$

The following proposition states that these estimates converge as desired.

Proposition 2.13. *Let q be a probability measure on a countable set Ω, and P_E be a probability measure on a subset $E \subseteq \Omega$ such that $q(E) > 0$ and $P_E(\omega)$ is proportional to $q(\omega)$ for $\omega \in E$. Suppose $s_1, s_2, \ldots$ is a sequence of outcomes sampled*

independently according to q. Then for any query event $Q \subseteq \Omega$, the estimates $\hat{p}_N$ defined in Equation 2.6 converge with probability one to $P_E(Q \cap E)$ as $N \to \infty$.

Proof. By the strong law of large numbers, $\sum_{n=1}^{N} \mathbf{1}_E(s_n)$ converges with probability one to $q(E)$. Since $q(E) > 0$, this implies that with probability one, this sum exceeds zero after some number of samples. After that point, $\hat{p}_N$ is given by the ratio in the second part of Equation 2.6. The numerator in this ratio converges with probability one to $q(Q \cap E)$, and the denominator converges with probability one to $q(E)$. Therefore the ratio converges with probability one to $\frac{q(Q \cap E)}{q(E)}$. We know $P_E(E) = 1$, so the proportionality constant β such that $P_E(\omega) = \beta q(\omega)$ must be $\frac{1}{q(E)}$. So $\frac{q(Q \cap E)}{q(E)} = P_E(Q \cap E)$. □

There is a simple method, sometimes called *logic sampling* [Henrion, 1988], for applying rejection sampling to any finite Bayesian network $\mathcal{B}$ that has a topological numbering. We let the sampling distribution q be the prior distribution $P_{\mathcal{B}}$ defined by the network. Assume that for each variable X in $\mathcal{B}$, we have a function `Sample` that takes in an instantiation of X's parents and returns a sample from X's CPD given these parent values. Then we can generate a sample from $P_{\mathcal{B}}$ by iterating over the variables in some topological order, and using `Sample` to sample a value for each one given the values already assigned to its parents. The result is a complete instantiation of the variables, which (in a discrete BN) corresponds to a single outcome. Given this procedure for sampling from $P_{\mathcal{B}}$, we can use the rejection sampling formula in Equation 2.6 to approximate $P_{\mathcal{B}}(Q|E)$.

In fact, we can remove the requirement that $\mathcal{B}$ be finite, as long as the evidence and query events are expressed as instantiations of finite sets of random variables. That is, $E = \mathrm{ev}(\mathbf{e})$ for some instantiation $\mathbf{e}$ of a finite set of *evidence variables* $\mathcal{V}_E$, and $Q = \mathrm{ev}(\mathbf{q})$ for some instantiation $\mathbf{q}$ of a finite set of *query variables* $\mathcal{V}_Q$. In this case, we can construct a BN $\mathcal{B}'$ from $\mathcal{B}$ by including just the query and evidence

variables and their ancestors: that is, the set of variables $\mathcal{V}' = \text{Anc}_{\mathcal{B}}(\mathcal{V}_E) \cup \text{Anc}_{\mathcal{B}}(\mathcal{V}_Q)$. It is clear by Lemma 2.7 that $\mathcal{B}'$ assigns the same probabilities to all instantiations of $\mathcal{V}'$ as $\mathcal{B}$ does. This observation is similar to Shachter's [1986] insight that *barren nodes* — that is, nodes that do not have any children — can be removed from a BN without affecting the joint distribution for the remaining variables.

Since we are assuming that $\mathcal{B}$ has a topological numbering, we know by Theorem 2.4 and Proposition 2.6 that each query or evidence variable has finitely many ancestors. Thus $\mathcal{V}'$ is finite. So we can apply rejection sampling to $\mathcal{B}'$ to compute an approximation to $P_{\mathcal{B}'}(\mathbf{q}|\mathbf{e})$; this result will also be an approximation to $P_{\mathcal{B}}(\mathbf{q}|\mathbf{e})$.

Thus, rejection sampling can be applied to any discrete BN that has a topological numbering. In principle, the estimates obtained from rejection sampling converge to the desired posterior probabilities. In practice, however, this convergence can be so slow as to make the algorithm useless. The problem is that the estimates only take into account samples that happen to land in E; the probability of this occurring on any given sample is $q(E)$. If our proposal distribution is the prior probability distribution for a BN $\mathcal{B}$, then $q(E)$ is the prior probability $P_{\mathcal{B}}(E)$. But in many cases, the probabilities of individual evidence instantiations decrease exponentially with the number of observed variables. For instance, if E represents an observed sequence of words that make up an English sentence, the probability of the sequence under the model can easily be 10^{-20} or smaller. This means that even if we are generating 100,000 samples per second, we will get one sample in E approximately every 30 million years.

2.5.2 Importance sampling and likelihood weighting

One way to achieve faster convergence than rejection sampling is to use a proposal distribution q that puts more probability mass on E. To create such a q that we

can sample from efficiently, we may need to sacrifice the property that $q(\omega)$ is proportional to $P_E(\omega)$ for $\omega \in E$: we may need to use a q that is biased toward certain portions of E. We can compensate for such bias by weighting the samples that we get, yielding an *importance sampling* algorithm [Rubinstein, 1981].

If we generate N samples $s_1, s_2, \ldots, s_N$ from q and use a weight function $w(\omega)$ that returns zero for $\omega \notin E$, we get a weighted estimate of $P_E(Q \cap E)$:

$$\hat{p}_N = \begin{cases} 0 & \text{if } \sum_{n=1}^{N} w(s_n) = 0 \\ \dfrac{\sum_{n=1}^{N} \mathbf{1}_{Q \cap E}(s_n) w(s_n)}{\sum_{n=1}^{N} w(s_n)} & \text{otherwise} \end{cases} \tag{2.7}$$

It turns out that to make $\hat{p}_N$ converge to $P_E(Q \cap E)$, it suffices to let $w(\omega)$ be proportional to $\frac{P_E(\omega)}{q(\omega)}$ for outcomes $\omega \in E$, and 0 on other outcomes. Rejection sampling is the special case where $w(\omega)$ is constant on E; the requirement that w yield zero on outcomes outside E means that those outcomes are effectively rejected.

Theorem 2.14. *Let q be a probability measure on a countable set Ω, and P_E be a probability measure on a subset $E \subseteq \Omega$ such that $q(E) > 0$. Let $w : \Omega \to \mathbb{R}$ be a function such that for some constant $\beta > 0$,*

$$w(\omega) = \begin{cases} \frac{\beta P_E(\omega)}{q(\omega)} & \textit{for } \omega \in E \\ 0 & \textit{otherwise.} \end{cases} \tag{2.8}$$

Suppose $s_1, s_2, \ldots$ is a sequence of outcomes sampled independently according to q. Then for any query event $Q \subseteq \Omega$, the estimates $\hat{p}_N$ defined in Equation 2.7 converge with probability one to $P_E(Q \cap E)$ as $N \to \infty$.

Proof. By the strong law of large numbers, $\sum_{n=1}^{N} w(s_n)$ converges with probability one to the expectation $\sum_{(\omega \in \Omega)} w(\omega) q(\omega)$. By Equation 2.8, this is equal to $\beta \sum_{(\omega \in E)} P_E(\omega)$, which is simply equal to β since P_E is a discrete probability mea-

sure on E. Since $\beta > 0$, it follows that with probability one, the second case in Equation 2.7 applies after some number of samples. In this case, $\hat{p}_N = \frac{\sum_{n=1}^{N} \mathbf{1}_{Q \cap E}(s_n) w(s_n)}{\sum_{n=1}^{N} w(s_n)}$. We already know that the denominator in this ratio converges with probability one to β. The numerator converges with probability one to $\sum_{(\omega \in \Omega)} \mathbf{1}_{Q \cap E}(\omega) w(\omega) q(\omega)$, which is equal to $\beta \sum_{(\omega \in Q \cap E)} P_E(\omega) = \beta P_E(Q \cap E)$. Therefore, the ratio converges with probability one to $P_E(Q \cap E)$. □

There is a simple and elegant way to apply importance sampling to a BN, called *likelihood weighting* [Fung and Chang, 1990; Shachter and Peot, 1990]. Consider a finite BN $\mathcal{B}$ over a set of variables $\mathcal{V}$, and assume it has a topological numbering. As above, we assume E corresponds to an instantiation $\mathbf{e}$ of some evidence variables $\mathcal{V}_E \subseteq \mathcal{V}$. In the likelihood weighting algorithm, we sample complete instantiations from the BN as described in the previous section, except that whenever we reach a variable $X \in \mathcal{V}_E$, we deterministically set it to its observed value $\mathbf{e}_X$. This means that every sample we generate is consistent with the evidence.

Clearly, we are no longer sampling from the prior distribution $P_{\mathcal{B}}$ with this algorithm. Instead, the probability of sampling an outcome ω includes factors for the unobserved variables only:

$$q(\omega) = \prod_{X \in \mathcal{V} \setminus \mathcal{V}_E} c_{\mathcal{B}}^X(X(\omega), \mathrm{Pa}_{\mathcal{B}}(X)(\omega))$$

We also know that by Lemma 2.7 (and the fact that in a discrete BN, there is a one-to-one correspondence between outcomes and complete instantiations):

$$P_E(\omega) = \frac{\prod_{X \in \mathcal{V}} c_{\mathcal{B}}^X(X(\omega), \mathrm{Pa}_{\mathcal{B}}(X)(\omega))}{P_{\mathcal{B}}(E)}$$

Therefore, for $\omega \in E$:

$$\begin{aligned}
\frac{P_E(\omega)}{q(\omega)} &= \frac{1}{P_{\mathcal{B}}(E)} \prod_{X \in \mathcal{V}_E} c_{\mathcal{B}}^X(X(\omega), \mathrm{Pa}_{\mathcal{B}}(X)(\omega)) \\
&= \frac{1}{P_{\mathcal{B}}(E)} \prod_{X \in \mathcal{V}_E} c_{\mathcal{B}}^X(\mathbf{e}_X, \mathrm{Pa}_{\mathcal{B}}(X)(\omega)) \\
\frac{P_{\mathcal{B}}(E)P_E(\omega)}{q(\omega)} &= \prod_{X \in \mathcal{V}_E} c_{\mathcal{B}}^X(\mathbf{e}_X, \mathrm{Pa}_{\mathcal{B}}(X)(\omega))
\end{aligned}$$

So the following weight function w satisfies Equation 2.8 with the proportionality constant $\beta = P_{\mathcal{B}}(E)$:

$$w(\omega) = \begin{cases} \prod_{X \in \mathcal{V}_E} c_{\mathcal{B}}^X(\mathbf{e}_X, \mathrm{Pa}_{\mathcal{B}}(X)(\omega)) & \text{if } X(\omega) = \mathbf{e}_X \text{ for all } X \in \mathcal{V}_E \\ 0 & \text{otherwise} \end{cases}$$

It is easy to compute this weight for an outcome ω as we are sampling it. We start with a weight of 1, and whenever we instantiate a variable $X \in \mathcal{V}_E$ to it observed value, we multiply its likelihood $c_{\mathcal{B}}^X(\mathbf{e}_X, \mathrm{Pa}_{\mathcal{B}}(X)(\omega))$ into the weight.

Thus, likelihood weighting is a form of importance sampling with a particular choice of q and w such that we can efficiently sample from q and compute w. Using the trick mentioned at the end of the previous section to deal with infinite networks, we can run likelihood weighting on any discrete BN that has a topological numbering. Convergence is often much faster than rejection sampling, since every sample is consistent with the evidence.

However, likelihood weighting becomes unacceptably slow to converge when there are tight dependencies between the evidence variables and their unobserved parents. For instance, consider a model with unobserved variables representing words in a sentence, and evidence variables representing the words written by an imperfect scribe trying to copy that sentence. When we generate a sample from q, we sample

the unobserved words from their prior distribution, and we are very unlikely to get a word sequence that is similar to the observed one. Thus, the likelihood weight — which is the probability of the observed words given the hidden ones — will be very small, or zero if the model assigns zero probability to certain copying errors. As a result, the estimates $\hat{p}_N$ end up having high variance unless N is extremely large: a handful of samples often get weights orders of magnitude larger than the others, and thus dominate the estimate [Russell and Norvig, 2003].

2.5.3 Markov chain Monte Carlo

So far we have discussed Monte Carlo algorithms in which the samples are drawn independently from some proposal distribution. This means that if we are lucky enough to generate a high-weight sample — one in which the values of the unobserved variables match the observed evidence well — we ignore that sample when generating the next one. It seems that it would be more efficient to let each sample depend on the preceding one, in such a way that we spend time exploring parts of the outcome space that have high posterior probability. *Markov chain Monte Carlo* (MCMC) algorithms [Metropolis *et al.*, 1953] are motivated by this intuition. In this section, we review certain aspects of MCMC algorithms that we will refer to later in the thesis; for a more thorough overview, see the book edited by Gilks *et al.* [1996] or the survey article by Andrieu *et al.* [2003].

A *Markov chain* on a countable set E is a sequence of random variables $S_0, S_1, S_2, \ldots$ taking values in E, with a joint distribution M that satisfies the *Markov property*: S_{n+1} may depend on S_n, but $S_{n+1} \perp\!\!\!\perp_M \{S_0, \ldots, S_{n-1}\} \mid S_n$. We will also assume that the chain is *homogeneous*: there is a fixed *transition distribution* M_{xy} such that $M(S_{n+1} = y | S_n = x) = M_{xy}$ for all natural numbers n and all $x, y \in E$. Intuitively, a Markov chain can be thought of as a random process that chooses a state S_0 ac-

cording to an initial state distribution M_0, then chooses a next state S_1 given S_0 according to M_{xy}, then chooses S_2 given S_1, and so on.

Our plan is to construct a Markov chain on an evidence event E such that if we sample outcomes $s_0, s_1, s_2, \ldots$ from the chain, then the estimates:

$$\hat{p}_N = \frac{1}{N} \sum_{n=1}^{N} \mathbf{1}_Q(s_n) \tag{2.9}$$

converge to $P_E(Q \cap E)$. In order for this to happen, P_E will need to be a *stationary distribution* of the chain.

Definition 2.20. *Let M_{xy} be a transition distribution on a countable set Ω. A probability measure π on E is a* stationary distribution *for M_{xy} if, for each $y \in E$,*

$$\sum_{x \in E} \pi(x) M_{xy} = \pi(y).$$

This definition says that a stationary distribution π is preserved by the transition distribution M_{xy}, in the following sense: if π is the marginal distribution of S_n, then the probability that S_{n+1} has a value y — which can be obtained by summing over all possible values x of S_n — is $\pi(y)$ as well.

The requirement that P_E be a stationary distribution for M_{xy} is not sufficient in itself to ensure that the estimates in Equation 2.9 converge to the desired probabilities. We also need to ensure that the chain does not get stuck in some subset B of the outcome space such that $P_E(B) < 1$. Let us write M^t_{xy} to denote $M(S_{n+t} = y | S_n = x)$; because of the Markov property and the assumption of homogeneity, this probability does not depend on n and is solely a property of the transition distribution. Following Tierney [1996], we say that a transition distribution M_{xy} is π-*irreducible* if for each pair $x, y \in E$ with $\pi(x) > 0$ and $\pi(y) > 0$, there

is some t such that $M^t_{xy} > 0$.[3] That is, within the set $\{x \in E : \pi(x) > 0\}$, there is a positive probability of getting from any outcome to any other outcome in some finite number of steps.

Theorem 2.15. *Let P_E be a probability distribution on a countable set E, and let $s_0, s_1, s_2, \ldots$ be a sequence of samples from a Markov chain with initial state distribution M_0 and transition distribution M_{xy}. Suppose P_E is a stationary distribution for M_{xy}, M_{xy} is P_E-irreducible, and $M_0(x) = 0$ for all $x \in E$ such that $P_E(x) = 0$. Then for any set Q, the estimates $\hat{p}_N$ defined in Equation 2.9 converge with probability one to $P_E(Q \cap E)$ as $N \to \infty$.*

The statement of this theorem is based on Theorem 4.3 of Tierney [1996]. It is a consequence of a more general result called the ergodic theorem (see, *e.g.*, Durrett [1996]). A Markov chain whose transition distribution satisfies the conditions of this theorem is said to be *ergodic*. Note that the distribution for the initial state S_0 is irrelevant; the only aspect of the Markov chain that matters is its transition distribution.

Thus, if we wish to use Markov chains for approximate inference, we need to generate samples from a Markov chain that has P_E as a stationary distribution. The *Metropolis-Hastings algorithm* [Metropolis *et al.*, 1953; Hastings, 1970] is a general technique for constucting such a chain. Like importance sampling, the Metropolis-Hastings algorithm allows us to use a *proposal distribution* q: but now q is a conditional distribution $q(s'|s)$.

For a given target distribution P_E, initial state distribution q_0, and proposal distribution q, the Metropolis-Hastings algorithm generates samples $s_0, s_1, s_2, \ldots$ by the following procedure:

[3] It is more common to define irreducibility without reference to any particular distribution, simply requiring that the property hold for all pairs of outcomes x, y. But if P_E assigns probability zero to some outcomes in E, then the Markov chains we construct for inference typically will not be irreducible in that sense: M^t_{xy} will be zero for all t if $P_E(y) = 0$.

1. Sample s_0 according to q_0.

2. To generate each sample s_{n+1}:

   (a) Sample a "proposed" state s' according to $q(\cdot|s_n)$.

   (b) Compute the *acceptance probability*:

$$\alpha(s_n, s') = \min\left(1, \frac{P_E(s')q(s_n|s')}{P_E(s_n)q(s'|s_n)}\right) \tag{2.10}$$

   (c) With probability $\alpha(s_n, s')$, accept the proposal: that is, let $s_{n+1} = s'$. Otherwise, reject the proposal and let $s_{n+1} = s_n$.

Note that the acceptance probability $\alpha(s_n, s')$ depends on the relative probabilities of s' and s_n under P_E, as well as the relative probabilities of the forward proposal $s_n \to s'$ and the reverse proposal $s' \to s_n$. Note that if s' has probability zero under P_E, or if the reverse proposal $s' \to s_n$ has zero probability under q, then the proposal will be rejected. The factor $q(s'|s_n)$ in the denominator will never be zero because s' was sampled from $q(\cdot|s_n)$. Also, as long as q_0 does not assign positive probability to states outside $\{s \in E : P_E(s) > 0\}$, we know the sampled states will all be in this set, so the factor $P_E(s_n)$ will not be zero either.

Theorem 2.16 (Hastings [1970]). *Let P_E be a probability measure on a countable set E, and let $q(s'|s)$ be any function such that for each $s \in E$, $q(\cdot|s)$ is a probability distribution on E. Then the Markov chain generated by the Metropolis-Hastings algorithm with target distribution P_E and proposal distribution q has P_E as a stationary distribution.*

If the resulting Markov chain is P_E-irreducible and $q_0(x) = 0$ whenever $P_E(x) = 0$, then by Theorem 2.15, the estimates $\hat{p}_N$ converge with probability one to $P_E(Q \cap E)$

as $N \to \infty$. Besides the requirement that the resulting chain be ergodic, no other conditions must be imposed on the proposal distribution q to ensure correctness.

However, the choice of proposal distribution can have a drastic effect on the speed of convergence. A good proposal distribution should propose outcomes that have relatively high probability given the evidence; if there are several regions in the outcome space that have high posterior probability, the proposal distribution should propose moves that go between them. Achieving these properties in practice can be difficult. There are some families of proposal distributions, such as Gibbs samplers [Geman and Geman, 1984], that can be applied to a wide variety of models. However, it is often necessary to develop a special-purpose proposal distribution to achieve fast convergence on a particular task [Tu and Zhu, 2002; Pasula *et al.*, 2003; Oh *et al.*, 2004]. The advantage of Metropolis-Hastings MCMC is that with a properly designed proposal distribution, it is sometimes possible to do inference on problems that would be intractable with all other known methods.

Chapter 3

Contingent Probabilistic Models

3.1 Motivation

In standard Bayesian networks (BNs), as discussed in Section 2.4, the dependency structure is fixed: each variable X either is or is not a parent of each variable Y. As we noted in the introduction, this limits the utility of BNs for scenarios where the relations between objects are unknown. Consider the following simple example from Russell [2001], which is a stylized version of the general problem of making inferences about the objects that underlie one's observations.

Example 3.1. *Suppose we have an urn that contains a finite but unknown number of balls; our prior distribution over the number of balls assigns positive probability to every natural number. Each ball has a color—say, blue or green—chosen independently from a fixed prior distribution. Suppose we repeatedly draw a ball uniformly at random, observe its color, and return it to the urn. We cannot distinguish two identically colored balls from each other. Furthermore, we have some (known) probability of making a mistake in each color observation. Given our observations, we might want to predict the total number of balls in the urn, or compute the posterior*

probability that we drew the same ball on our first two draws.

It is natural to think of the possible outcomes of this scenario as *logical model structures* (defined in Section 2.1) that contain balls, draws and colors and specify certain relations between them. Specifically, we will use a typed first-order language with three types: Ball, Draw and Color. The language includes constant symbols Blue and Green for the colors, and constant symbols of the form Draw1, Draw2, etc. for the draws (we know how many of these symbols to include because we know how many draws we are making). The function symbols of the language are $\mathsf{TrueColor} : \mathsf{Ball} \rightarrow \mathsf{Color}$, $\mathsf{BallDrawn} : \mathsf{Draw} \rightarrow \mathsf{Ball}$, and $\mathsf{ObsColor} : \mathsf{Draw} \rightarrow \mathsf{Color}$. For a scenario with k draws, we will let our outcome space Ω be the set of model structures ω of this language in which $[\mathsf{Color}]^\omega = \{\mathsf{Blue}, \mathsf{Green}\}$, $[\mathsf{Draw}]^\omega$ is the set of pairs $\{(\mathsf{Draw}, 1), \ldots, (\mathsf{Draw}, k)\}$, and $[\mathsf{Ball}]^\omega = \{(\mathsf{Ball}, 1), \ldots, (\mathsf{Ball}, n)\}$ for some natural number n. We also restrict Ω to model structures where the constant symbols have the obvious intended interpretations, so only the interpretations of TrueColor, BallDrawn, and ObsColor can vary. Each of these outcomes contains finitely many objects, so Ω is countable, and we can let the event space $\mathscr{F}$ be the power set of Ω.

A natural method for defining a probability measure over $(\Omega, \mathscr{F})$ is to define some random variables on the outcome space and specify a BN over these random variables. Figure 3.1 shows what appears to be a natural BN for an urn-and-balls scenario with two draws. For a scenario with k draws, the random variables are N, $\{C_i\}_{i=1}^{\infty}$, $\{B_j\}_{j=1}^{k}$ and $\{O_j\}_{j=1}^{k}$, defined as follows:

$$
\begin{aligned}
N(\omega) &= |\,[\mathsf{Ball}]^\omega\,| \\
C_i(\omega) &= \begin{cases} [\mathsf{TrueColor}]^\omega\,((\mathsf{Ball}, i)) & \text{if } i \leq |\,[\mathsf{Ball}]^\omega\,| \\ \mathsf{null} & \text{otherwise} \end{cases} \\
B_j(\omega) &= [\mathsf{BallDrawn}]^\omega\,((\mathsf{Draw}, j)) \\
O_j(\omega) &= [\mathsf{ObsColor}]^\omega\,((\mathsf{Draw}, j))
\end{aligned}
$$

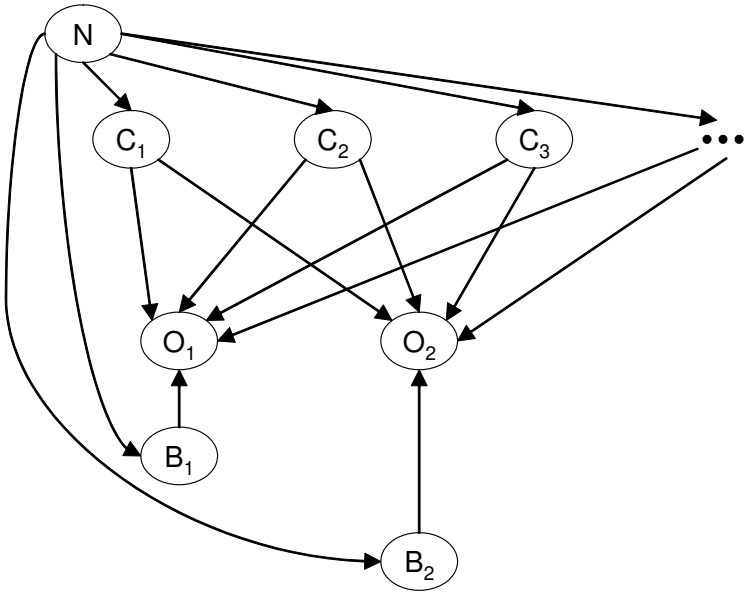


Figure 3.1: A graphical model for the balls-and-urn example with two draws. Because the "observed color" nodes O_1 and O_2 have infinitely many parents, this model is not a Bayesian network under the definition we gave in Section 2.4.

However, there are several difficulties with the model in Figure 3.1. First, the set of outcomes we have defined is not isomorphic to the Cartesian product of the ranges of our random variables. Some instantiations of the random variables are *unachievable*: they do not correspond to any outcome. For instance, there are no outcomes ω where $N(\omega) = 5$ and $C_{20}(\omega) = \mathsf{Blue}$. Thus, the model in Figure 3.1 does not satisfy the conditions of Definition 2.18. Some choices of CPDs for the variables will fail to yield a probability measure over $(\Omega, \mathscr{F})$, because some probability mass will be placed on unachievable instantiations.

The model in Figure 3.1 also violates the requirement (in Definition 2.17) that every variable have finitely many parents. Because the observed color on a draw may depend on the true color of any ball, the $\mathsf{ObsColor}$ variables $O_1, \ldots, O_k$ have incoming edges from all of the $\mathsf{TrueColor}$ variables $C_1, C_2, \ldots$. Note that we cannot limit ourselves to a finite set of $\mathsf{TrueColor}$ variables $C_1, \ldots, C_m$ because then our

model would not specify probabilities for outcomes with more than m balls. It would be possible to add intermediate, deterministic variables to the model so that it represented the same probability distribution using only finite parent sets. However, the ObsColor variables would still end up with infinitely many ancestors. So we could not use the results of Section 2.4 to prove that the BN defines a unique probability measure.

One could argue that infinite sets of variables are a purely theoretical concern: in practice, we could limit the number of balls to, say, 10,000. But still, we would be left with a very large BN that would create difficulties for standard inference algorithms. Note that the standard technique of reducing the size of a BN by removing "barren" nodes — that is, nodes that are not ancestors of any query or evidence nodes [Shachter, 1986] — is not helpful here, because $C_1, C_2, \ldots$ are all ancestors of $O_1, \ldots, O_k$, which are observed. The problem is that the BN does not make explicit the contingent nature of the dependencies between the ObsColor and TrueColor variables. As we will show in Chapter 5, an algorithm that exploits contingent dependencies can perform inference without assuming any upper bound on the number of balls.

We have seen that for scenarios with unknown objects and relational uncertainty, the natural graphical models may not satisfy the requirement that each node have finitely many ancestors. Relational uncertainty can also lead to models that violate the other well-definedness condition we gave in Section 2.4, namely acyclicity. We will see this in the next example.

Example 3.2. *Suppose a hurricane is going to strike two cities, Alphatown and Betaville, but it is not known which city will be hit first. The amount of damage in each city depends on the level of preparations made in each city. Also, the level of preparations in the second city to be hit depends on the amount of damage in the*

first city.

Because this example involves no uncertainty about what objects exist, it would be straightforward to represent the possible outcomes simply as instantiations of a set of random variables. However, for consistency, we will continue to represent the outcomes as logical model structures. We can represent this scenario using a logical language with types City, PrepLevel (for the level of preparations) and DamageLevel. Each of these types has a known extension: in every possible world ω, $[\mathsf{City}]^\omega = \{\mathsf{A}, \mathsf{B}\}$ (standing for Alphatown and Betaville); $[\mathsf{PrepLevel}]^\omega = \{\mathsf{High}, \mathsf{Low}\}$; and $[\mathsf{DamageLevel}]^\omega = \{\mathsf{Mild}, \mathsf{Severe}\}$. In addition to nonrandom constant symbols denoting these known objects, we include a constant symbol First whose value is the City object that is hit first by the hurricane. The value of First varies from outcome to outcome. Thus, we have a simple kind of relational uncertainty: we do not know which city plays the role of the first city to be hit. Finally, we have two unary function symbols: $\mathsf{Prep} : \mathsf{City} \rightarrow \mathsf{PrepLevel}$ and $\mathsf{Damage} : \mathsf{City} \rightarrow \mathsf{DamageLevel}$. On this set of model structures, we define random variables F, $\{P_c\}_{c \in \{A,B\}}$ and $\{D_c\}_{c \in \{A,B\}}$ as follows:

$$\begin{aligned} F(\omega) &= [\mathsf{First}]^\omega \\ P_c(\omega) &= [\mathsf{Prep}]^\omega (c) \\ D_c(\omega) &= [\mathsf{Damage}]^\omega (c) \end{aligned}$$

In this example, suppose that we have a good estimate of the distribution for preparations in the first city, and of the conditional probability distribution for preparations in the second city given damage in the first. The obvious graphical model to draw is the one in Fig. 3.2, but it has a figure-eight-shaped cycle. Of course, we can construct a BN for the intended distribution by choosing an arbitrary ordering of the variables and including all necessary edges to each variable from its predeces-

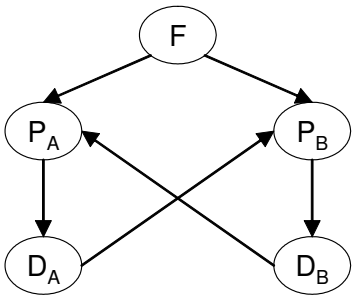


Figure 3.2: A cyclic graphical model for the hurricane scenario. P stands for preparations, D for damage, A for Alphatown, B for Betaville, and F for the city that is hit first.

sors. Suppose we use the ordering F, P_A, D_A, P_B, D_B. Then $P(P_A|F = A)$ is easy to write down, but to compute $P(P_A|F = B)$ we need to sum out P_B and D_B. There is no acyclic BN that reflects our causal intuitions. And recall that in general, as we showed in Section 2.4.3, a cyclic BN may not be satisfied by any probability measure.

Note that in this example, the damage in Alphatown influences the preparations in Betaville only when Alphatown is the first city to be hit. Thus, the edge from D_A to P_B is active only when $F = A$; similarly, the edge from D_B to P_A is active only when $F = B$. So there is no context in which all the edges in a cycle are active. Similarly, in the urn-and-balls example, if we condition on $B_k = i$, then O_k has only one other ancestor: C_i. Thus, the ancestor sets are "context-specifically" finite.

In this chapter, we develop modeling formalisms that explicitly represent the conditions under which dependencies are active. This idea of representing *context-specific independence* has been explored before [Boutilier *et al.*, 1996] with the goal of obtaining more compact representations and performing inference more efficiently. We go beyond this earlier work in that we provide new criteria for showing that a model defines a unique probability measure; these criteria can be satisfied even for models whose BN representations would contain cycles or infinite ancestor sets.

We begin in Section 3.2 by addressing the first problem we mentioned with Ex-

ample 3.1: that the outcome space is not the product of the ranges of the random variables. We give weaker conditions under which a set of random variables can be used to define a probability measure over an outcome space. In Section 3.3, we lay the foundation for models with contingent dependencies by giving an alternative version of Kolmogrov's extension theorem. The version we gave in Section 2.2.5 requires a fixed ordering of the random variables $X_0, X_1, \ldots$, and requires probabilities to be specified for all instantiations of all finite prefixes of this ordering. Our new version of the theorem only requires probabilities to be specified for all the instantiations in a *split tree*, which defines a contingent ordering on the variables. Split trees serve as the semantic foundation for *partition-based models* (PBMs), introduced in Section 3.4. In a PBM, instead of specifying a set of parents for each variable, one specifies an arbitrary partition of the outcome space that determines the variable's conditional distribution. We identify a condition under which a PBM is guaranteed to define a unique probability measure. In the special case where the partition for each variable is defined by a decision tree, a PBM can be represented as a *contingent Bayesian network* (CBN): a directed graph where edges are labeled with the conditions under which they are active. In Section 3.5, we provide graphical conditions under which a CBN defines a unique probability distribution. These conditions may be satisfied even in the presence of cycles or infinite ancestor sets.

3.2 Non-product outcome spaces

Suppose we are given a measurable space $(\Omega, \mathscr{F})$ and a countable set of random variables $\mathcal{V}$. Throughout this chapter, we assume the random variables in $\mathcal{V}$ are *discrete*: each variable X takes values in a value space $(S_X, \mathscr{S}_X)$ where S_X is countable and $\mathscr{S}_X$ is the power set of S_X. As usual, we write $\text{range}(X)$ to denote S_X. Note that although $\mathcal{V}$ is countable and each variable has a countable range, the number of

complete instantiations of these variables is uncountable (unless $\mathcal{V}$ is finite). For instance, the instantiations of a countable sequence of binary variables can be thought of as infinite binary expansions of the real numbers.

As we noted in the urn-and-balls example (Example 3.1), some instantiations of the random variables may not correspond to any outcome. That is, they may not be *achievable*.

Definition 3.1. *Let $\mathcal{V}$ be a set of random variables defined on a measurable space $(\Omega, \mathscr{F})$. An instantiation σ on $\mathcal{V}$ is* achievable *if* $\mathrm{ev}(\sigma) \neq \varnothing$.

Although the complete instantiations of $\mathcal{V}$ in Example 3.1 are not in one-to-one correspondence with the outcomes, an achievable complete instantiation does fully describe an outcome. The values of the variables $\{C_i\}_{i=1}^{\infty}$, $\{B_j\}_{j=1}^{k}$ and $\{O_j\}_{j=1}^{k}$ specify the interpretations of TrueColor, BallDrawn and ObsColor, respectively, and N specifies the extension of the type Ball. All other aspects of the model structures in Ω are fixed. Thus, we can fully specify a probability measure over $(\Omega, \mathscr{F})$ by specifying a probability measure over the achievable instantiations of $\mathcal{V}$.

To formalize this idea, we will prove a version of Kolmogorov's extension theorem (Theorem 2.2) that allows non-product outcome spaces. This theorem will require that $\mathcal{V}$ be *sufficient* for $(\Omega, \mathscr{F})$ in the following sense.

Definition 3.2. *A set $\mathcal{V}$ of discrete random variables is* sufficient *for a measurable space $(\Omega, \mathscr{F})$ if:*

i. the events of the form $\{X = x\}$ for $X \in \mathcal{V}$ and $x \in \mathrm{range}(X)$ generate $\mathscr{F}$; and

ii. if σ is a complete instantiation on $\mathcal{V}$, and every finite sub-instantiation of σ is achievable, then σ is achievable.

Condition (i) says, intuitively, that all measurable events can be expressed in terms of assignments of values to the random variables. This ensures that a joint

distribution for $\mathcal{V}$ corresponds to a unique distribution on $(\Omega, \mathscr{F})$. Condition (ii) says that we can determine if a complete instantiation σ is achievable or not by looking at its finite sub-instantiations. This condition is perhaps easier to check in its contrapositive form: every unachievable complete instantiation has an unachievable finite sub-instantiation. In Example 3.1, the only way a complete instantiation σ can be unachievable is if there is some $i > \sigma_N$ such that $\sigma_{(C_i)} \neq \mathsf{null}$ for some $i > \sigma_N$. But then the finite sub-instantiation consisting of N and this C_i is also unachievable, so condition (ii) is satisfied. Note that the restriction to complete instantiations is important here. An incomplete instantiation that does not instantiate N, but asserts $C_i = \mathsf{Blue}$ for $i = 1$ to ∞, is unachievable because every model structure in Ω contains a finite number of balls.

To get an idea of what outcome spaces we are ruling out with condition (ii), consider a scenario where each outcome $\omega \in \Omega$ is an infinite sequence $(\omega_i)_{i=1}^{\infty}$, with $\omega_i \in \{0, 1\}$ for each i. Let $\mathscr{F}$ be generated by the events of the form $\{\omega \in \Omega : \omega_i = 1\}$ for $i = 1$ to ∞. If we let $\mathcal{V} = \{X_i\}_{i=1}^{\infty}$ where $X_i(\omega) = \omega_i$, then condition (i) is satisfied. Now suppose Ω consists of just those sequences that contain finitely many 1's. If σ is a complete instantiation that sets an infinite set of X_i variables to 1, then it is unachievable, although each of its finite sub-instantiations is achievable. So condition (ii) is not satisfied in this case.

We now give a version of Kolmogorov's extension theorem that does not require the random variables to be the coordinate variables of a product space. Instead, we just require that the random variables be sufficient for the outcome space on which they are defined. The only other difference between this theorem and Theorem 2.2 is in the consistency condition for the finite-dimensional distribution functions f_n, which now forces unachievable instantiations to have zero probability. This condition uses a new piece of notation: for an event $A \subseteq \Omega$, we will write $\text{range}\,(X|A)$ to denote $\{x \in \text{range}\,(X) : \exists \omega \in A\,(X(\omega) = x)\}$.

Theorem 3.1. *Let $\mathcal{V}$ be a set of discrete random variables that are sufficient for a measurable space $(\Omega, \mathscr{F})$. Let $X_0, X_1, \ldots$ be any numbering of $\mathcal{V}$. For each natural number $n \leq |\mathcal{V}|$, let f_n be a function from $\text{range}(\{X_0, \ldots, X_{n-1}\})$ to $[0,1]$. Suppose that $f_0(\top) = 1$, and for all natural numbers $n < |\mathcal{V}|$ and all instantiations $\sigma \in \text{range}(\{X_0, \ldots, X_{n-1}\})$,*

$$\sum_{x_n \in \text{range}(X_n|\sigma)} f_{n+1}(\sigma; X_n = x_n) = f_n(\sigma) \tag{3.1}$$

Then there is a unique probability measure P on $(\Omega, \mathscr{F})$ such that $P(\sigma) = f_n(\sigma)$ for each natural number $n \leq |\mathcal{V}|$ and each $\sigma \in \text{range}(\{X_0, \ldots, X_{n-1}\})$.

Proof. Suppose each variable $X \in \mathcal{V}$ has a value space $(S_X, \mathscr{S}_X)$, and let $(\Omega_\mathcal{V}, \mathscr{F}_\mathcal{V})$ be the product space $\times_{(X \in \mathcal{V})}(S_X, \mathscr{S}_X)$. Note that $\Omega_\mathcal{V}$ is the set of complete instantiations of $\mathcal{V}$ (see Section 2.2.4). Now, for each $X \in \mathcal{V}$, we define a random variable $\widetilde{X} : (\Omega_\mathcal{V}, \mathscr{F}_\mathcal{V}) \to (S_X, \mathscr{S}_X)$, such that $\widetilde{X}(\sigma) \triangleq \sigma_X$. These variables, which we will denote collectively as $\widetilde{\mathcal{V}}$, are the coordinate variables of $\Omega_\mathcal{V}$. They can be numbered $\widetilde{X}_0, \widetilde{X}_1, \ldots$ in correspondence with $X_1, X_2, \ldots$. For any given instantiation σ on $\mathcal{V}$, we will write $\widetilde{\sigma}$ for the instantiation on $\widetilde{\mathcal{V}}$ such that $\text{vars}(\widetilde{\sigma}) = \{\widetilde{X} : X \in \text{vars}(\sigma)\}$ and $\widetilde{\sigma}_{\widetilde{X}} = \sigma_X$ for each $\widetilde{X} \in \text{vars}(\widetilde{\sigma})$. Note that $\text{ev}(\widetilde{\sigma})$ consists of all those complete instantiations that are extensions of σ. Now for each $n \leq |\mathcal{V}|$, let $\widetilde{f}_n$ be a function on $\text{range}\left(\{\widetilde{X}_0, \ldots, \widetilde{X}_{n-1}\}\right)$ such that $\widetilde{f}_n(\tilde{\sigma}) = f_n(\sigma)$. By the standard version of Kolmogorov's extension theorem (Theorem 2.2), there is a unique probability measure $P_\mathcal{V}$ on $(\Omega_\mathcal{V}, \mathscr{F}_\mathcal{V})$ such that $P_\mathcal{V}(\widetilde{\sigma}) = \widetilde{f}_n(\widetilde{\sigma})$ for each $n \leq |\mathcal{V}|$ and $\widetilde{\sigma} \in \text{range}\left(\{\widetilde{X}_0, \ldots, \widetilde{X}_{n-1}\}\right)$.

Now let V be a function from $(\Omega, \mathscr{F})$ to $(\Omega_\mathcal{V}, \mathscr{F}_\mathcal{V})$ such that $V(\omega)$ is the instantiation that assigns the value $X(\omega)$ to each $X \in \mathcal{V}$. Because each element of $\mathcal{V}$ is a random variable and $(\Omega_\mathcal{V}, \mathscr{F}_\mathcal{V})$ is a product space, V is also a random variable. By the definitions of $\widetilde{X}$ and $\widetilde{\sigma}$, we know $V^{-1}(\text{ev}(\widetilde{\sigma})) = \text{ev}(\sigma)$ for each instantiation $\widetilde{\sigma}$

on $\widetilde{\mathcal{V}}$. And for each $n \leq |\mathcal{V}|$, the instantiations of $\{\widetilde{X}_0, \ldots, \widetilde{X}_{n-1}\}$ are in one-to-one correspondence with the instantiations of $\{X_0, \ldots, X_{n-1}\}$. So a probability measure P on $(\Omega, \mathscr{F})$ satisfies the conditions of this proposition if and only if $P_{\mathcal{V}}$ is the distribution on V induced by P.

We will use Prop. 2.3 to show that there is a unique probability measure that induces $P_{\mathcal{V}}$. The first thing to show is that the preimages (under V) of events in $\mathscr{F}_{\mathcal{V}}$ generate $\mathscr{F}$. Since $\mathcal{V}$ is sufficient for $(\Omega, \mathscr{F})$, we know the events of the form $\{X = x\}$ for $X \in \mathcal{V}$ generate $\mathscr{F}$. These are the preimages of events of the form $\{\sigma_X = x\}$, which are in $\mathscr{F}_{\mathcal{V}}$ since the variables are discrete. So condition (i) of Prop. 2.3 is satisfied.

The other thing we have to show is that $P_{\mathcal{V}}$ assigns probability one to the set of achievable instantiations. First, for each natural number $n \leq |\mathcal{V}|$, let S_n be the set of achievable instantiations of $\{X_0, \ldots, X_{n-1}\}$. We will show by induction on n that $\sum_{(\sigma \in S_n)} P_{\mathcal{V}}(\widetilde{\sigma}) = 1$. For $n = 0$, we just have $S_0 = \{\top\}$, and $P_{\mathcal{V}}(\widetilde{\top}) = \widetilde{f}(\widetilde{\top}) = f(\top) = 1$. Now suppose $\sum_{(\sigma \in S_n)} P_{\mathcal{V}}(\widetilde{\sigma}) = 1$, and consider S_{n+1}. Each instantiation $\sigma \in S_{n+1}$ has the form $(\tau; X_n = x_n)$ for some $\tau \in S_n$ and $x_n \in \text{range}\,(X_n|\tau)$. Therefore:

$$\sum_{\sigma \in S_{n+1}} P_{\mathcal{V}}(\widetilde{\sigma}) = \sum_{(\tau \in S_n)} \sum_{(x_n \in \text{range}(X_n|\tau))} P_{\mathcal{V}}(\widetilde{\tau}; \widetilde{X}_n = x_n)$$

But by Equation 3.1 and the definitions of $P_{\mathcal{V}}$ and $\widetilde{f}$, $\sum_{(x_n \in \text{range}(X_n|\tau))} P_{\mathcal{V}}(\widetilde{\tau}; \widetilde{X}_n = x_n) = P_{\mathcal{V}}(\widetilde{\tau})$. So we have $\sum_{(\sigma \in S_{n+1})} P_{\mathcal{V}}(\widetilde{\sigma}) = \sum_{(\tau \in S_n)} P_{\mathcal{V}}(\widetilde{\tau})$, which equals one by the inductive hypothesis.

Since $\text{ev}\,(\widetilde{\sigma})$ is the set of complete extensions of σ, $\bigcup_{(\sigma \in S_n)} \text{ev}\,(\widetilde{\sigma})$ is the set of all complete instantiations σ of $\mathcal{V}$ such that $\sigma[\{X_0, \ldots, X_{n-1}\}]$ is achievable. Let A_n denote this set of complete instantiations. Note that each A_n is $\mathscr{F}_{\mathcal{V}}$–measurable, since each event $\text{ev}\,(\widetilde{\sigma})$ is $\mathscr{F}_{\mathcal{V}}$–measurable and A_n is a countable union of such events. Also, by Definition 3.2(ii), we know that if every finite subinstantiation of σ is achievable then so is σ. So the set of all achievable instantiations of $\mathcal{V}$, which we will denote A,

is equal to $\bigcap_{n=0}^{|\mathcal{V}|} A_n$. Since A is a countable intersection of $\mathscr{F}_{\mathcal{V}}$–measurable sets that have probability one under $P_{\mathcal{V}}$, we know A itself is $\mathscr{F}_{\mathcal{V}}$–measurable and $P_{\mathcal{V}}(A) = 1$. So condition (ii) in Prop. 2.3 is satisfied, and P is the unique probability measure on $(\Omega, \mathscr{F})$ that induces $P_{\mathcal{V}}$. □

This version of Kolmogorov's extension theorem can be used to prove well-definedness results for BNs on non-product outcome spaces. However, we still have the requirement that such a BN must be acyclic and each of its variables must have finitely many ancestors. Thus, we need to go further to handle cases such as the urn-and-balls scenario of Example 3.1 or the hurricane scenario of Example 3.2.

3.3 Split trees

The versions of Kolmogorov's extension theorem that we have given so far require a numbering $X_0, X_1, \ldots$ of the random variables $\mathcal{V}$, and require probabilities to be specified for all instantiations of prefixes of this numbering. These conditions are already easier to work with than those in other versions of the extension theorem (see, *e.g.*, Billingsley [1995]) that require probabilities for all instantiations of *all* finite subsets of $\mathcal{V}$. In this section, we prove a result that allows still more flexibility in choosing the set of instantiations for which we specify probabilities. This result is based on the notion of a *split tree*, which can be thought of as a contingent numbering of the random variables. We will write $\mathrm{ch}_T(\sigma)$ to denote the children of a node σ in a rooted tree T.

Definition 3.3. *A* split tree *for a set of random variables $\mathcal{V}$ is a rooted tree T whose nodes are partial instantiations of $\mathcal{V}$, in which:*

i. the root node is the empty instantiation;

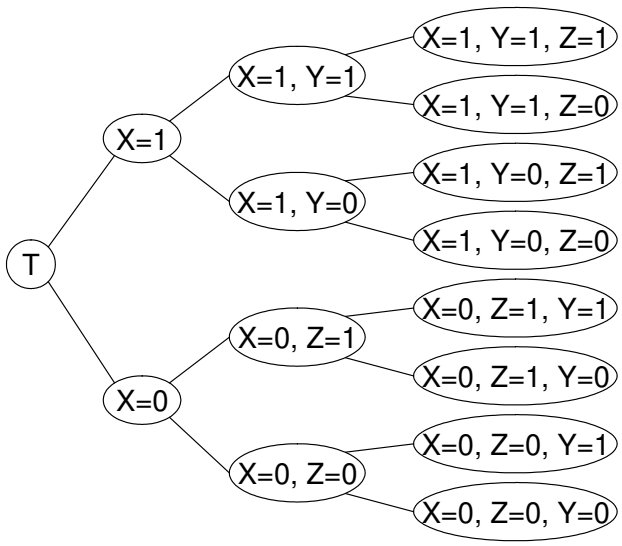


Figure 3.3: A split tree over the variables $\{X, Y, Z\}$.

ii. for each non-leaf instantiation σ in T, there is a split variable X_T^σ *such that*

$$\mathrm{ch}_T(\sigma) = \{(\sigma; X_T^\sigma = x) \,:\, x \in \mathrm{range}(X_T^\sigma|\sigma)\};$$

iii. for each non-truncated path (i.e., a path that does not end except at a leaf) starting at the root of T, and for each $X \in \mathcal{V}$, there is exactly one node σ on the path such that $X_T^\sigma = X$.

Figure 3.3 shows an example of a split tree. A split tree T can be thought of as a program for sampling a complete instantiation. We start with an empty instantiation. When we are at a non-leaf node σ, we stochastically choose one of σ's children, which corresponds to assigning some value to X_T^σ (the split tree does not define the probabilities of choices; those will be specified separately). Part (iii) of the definition ensures that we sample a value for each variable exactly once in each "run" of the program. Note that each instantiation in a split tree is finite, and since we are assuming that the ranges of random variables are countable, this implies that

any split tree contains at most countably many instantiations.

Our first result about split trees is that every instantiation in a split tree is achievable.

Lemma 3.2. *If T is a split tree for a set of variables $\mathcal{V}$ defined on a measurable space $(\Omega, \mathscr{F})$, then every instantiation in T is achievable.*

Proof. Consider any instantiation σ in T. If σ is the root node, then $\sigma = \top$, which is achievable since Ω is non-empty (this was part of our definition of a measurable space in Section 2.2.1). Otherwise, σ has a parent τ. By part (ii) of the definition of a split tree, $\sigma = (\tau; X = x)$ for some $x \in \text{range}(X|\tau)$. By the definition of $\text{range}(X|\tau)$, this implies there is some $\omega \in \text{ev}(\tau)$ such that $X(\omega) = x$. This ω is consistent with σ, so σ is achievable. □

We can also show that if $\mathcal{V}$ is sufficient for $(\Omega, \mathscr{F})$, then the conjunction of the instantiations along any non-truncated path from the root is achievable. This holds even if the path is infinite, in which case the conjunction is an infinite instantiation that is not itself in the tree.

Lemma 3.3. *Let T be a split tree for a set of variables $\mathcal{V}$ that is sufficient for $(\Omega, \mathscr{F})$. If $\sigma_1, \sigma_2, \ldots$ is a non-truncated path starting at the root of T, then the conjunction $\wedge_i \sigma_i$ is achievable.*

Proof. First, the conjunction $\wedge_i \sigma_i$ is well-defined because, by Definition 3.3(iii), each path splits on any given variable at most once; thus the instantiations $\sigma_1, \sigma_2, \ldots$ do not assign conflicting values to any variable. We now claim that $\wedge_i \sigma_i$ is a complete instantiation of $\mathcal{V}$. This is also implied by Definition 3.3(iii): for every $X \in \mathcal{V}$, there is exactly one node σ_j on this non-truncated path that splits on X. Thus the path contains a node σ_{j+1} that is a child of σ_j and hence instantiates X. So $\wedge_i \sigma_i$ assigns a value to every variable in $\mathcal{V}$.

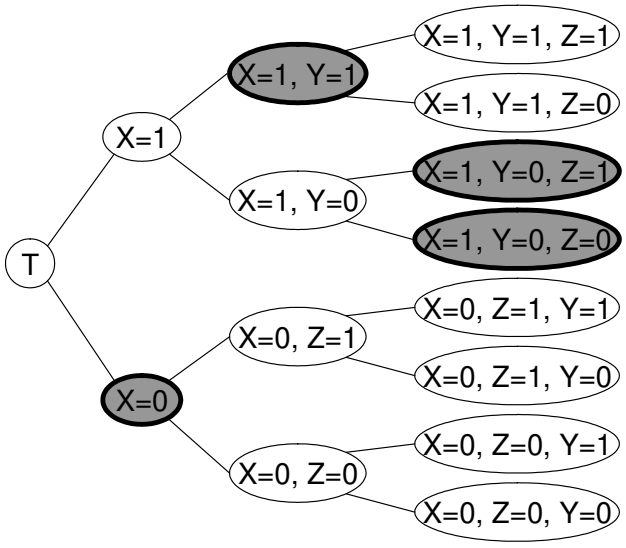


Figure 3.4: There is one shaded instantiation on each non-truncated path from the root node. By Lemma 3.4, the events corresponding to these instantiations form a partition of $\mathrm{ev}(\top)$.

Now let τ be any finite sub-instantiation of $\wedge_i \sigma_i$. For each variable $X \in \mathrm{vars}(\tau)$, there must be some j such that σ_j instantiates X. Let j_X be the least such j for any given X. Then let $j_\tau = \max_{(X \in \mathrm{vars}(\tau))} j_X$; this maximization is well-defined since τ is finite. It follows that τ is a sub-instantiation of $\sigma_{(j_\tau)}$. We know by Lemma 3.2 that $\sigma_{(j_\tau)}$ is achievable, so τ is achievable. Since this holds for any finite sub-instantiation of $\wedge_i \sigma_i$, the fact that $\mathcal{V}$ is sufficient for $(\Omega, \mathscr{F})$ implies that $\wedge_i \sigma_i$ is achievable. □

It is clear from Definition 3.3(ii) that in a split tree, the children of any node σ represent a partition of $\mathrm{ev}(\sigma)$. It is slightly less obvious that the same partition property holds if we choose one instantiation from each non-truncated path starting at σ. For example, if we choose one instantiation from each non-truncated path out of the root node — as shown in Figure 3.4 — then the events corresponding to these instantiations form a partition of $\mathrm{ev}(\top)$. The following lemma states this fact, which is useful in subsequent proofs.

Lemma 3.4. *In a split tree T, if R is a set of descendants of σ such that every non-truncated path starting from σ contains exactly one element of R, then $\{\mathrm{ev}(\rho) : \rho \in R\}$ is a partition of $\mathrm{ev}(\sigma)$.*

Proof. First, note that each event in $\{\mathrm{ev}(\rho) : \rho \in R\}$ is non-empty, since every instantiation in a split tree is achievable (by Lemma 3.2). Now consider any two distinct instantiations $\rho_1, \rho_2 \in R$. By hypothesis, ρ_1 and ρ_2 are on different paths from σ; let τ be the last node that is on both of these paths. The instantiations ρ_1 and ρ_2 must assign different values to X_T^τ, so they correspond to disjoint sets.

Now we will show that $\bigcup_{(\rho \in R)} \mathrm{ev}(\rho) = \mathrm{ev}(\sigma)$. Based on Definition 3.3(ii), it is clear that all descendants of σ represent subsets of $\mathrm{ev}(\sigma)$. So $\bigcup_{(\rho \in R)} \mathrm{ev}(\rho) \subseteq \mathrm{ev}(\sigma)$. To show $\mathrm{ev}(\sigma) \subseteq \bigcup_{(\rho \in R)} \mathrm{ev}(\rho)$, consider any outcome $\omega \in \mathrm{ev}(\sigma)$. We will construct a non-truncated path $\sigma_1, \sigma_2, \ldots$, starting at σ, on which every node is consistent with ω. Because $\omega \in \sigma$ and $\sigma_1 = \sigma$, we know σ_1 is consistent with ω. Now as an inductive hypothesis, suppose $\sigma_1, \sigma_2, \ldots, \sigma_n$ are all consistent with ω. If σ_n is a leaf node, then $\sigma_1, \sigma_2, \ldots, \sigma_n$ constitutes a non-truncated path. Otherwise, we can extend the path with $\sigma_{n+1} = (\sigma_n; X_T^{\sigma_n} = X_T^{\sigma_n}(\omega))$, which is a child of σ_n (by Definition 3.3(ii)) and is consistent with ω. In this way, we can construct a non-truncated path starting at σ on which all nodes are consistent with ω. Let ρ be the element of R on this path; then $\omega \in \mathrm{ev}(\rho)$. So every $\omega \in \mathrm{ev}(\sigma)$ is in some element of R. □

The next thing we would like to show about split trees is that if $\mathcal{V}$ is sufficient for $(\Omega, \mathscr{F})$, then the nodes in a split tree for $\mathcal{V}$ generate $(\Omega, \mathscr{F})$. This is not completely obvious, because the split tree does not contain instantiations of the form $(X = x)$ for most variables — indeed, the only variable for which such instantiations occur is the split variable of the root node. However, it turns out that we can express any set of the form $\{X = x\}$ as the union of instantiations in the tree.

Lemma 3.5. *Let T be a split tree for a set of variables $\mathcal{V}$ that is sufficient for $(\Omega, \mathscr{F})$. Then the set of events $\{\mathrm{ev}(\sigma) : \sigma \in T\}$ generates $\mathscr{F}$.*

Proof. Because $\mathcal{V}$ is sufficient for $(\Omega, \mathscr{F})$, it suffices to show that $\{\mathrm{ev}(\sigma) : \sigma \in T\}$ generates the set of events $\{X = x\}$ for $X \in \mathcal{V}$ and $x \in \mathrm{range}(X)$. So consider any such event. Let R be the set of nodes ρ in T such that $X_T^\rho = X$. By Definition 3.3(iii), there is exactly one element of R on each non-truncated path starting at the root. So by Lemma 3.4, $\bigcup_{(\rho \in R)} \mathrm{ev}(\rho) = \Omega$. Now let $R' = \{(\rho; X = x) : \rho \in R \text{ and } x \in \mathrm{range}(X|\rho)\}$. Note that by Definition 3.3(ii), all the instantiations in R' are in the tree. We claim that $\bigcup_{(\rho' \in R')} \mathrm{ev}(\rho') = \{X = x\}$. To see this, first note that the elements of R' all correspond to subsets of $\{X = x\}$ because they all instantiate X to x. To see that they are exhaustive, consider any ω such that $X(\omega) = x$. Because $\bigcup_{(\rho \in R)} \mathrm{ev}(\rho) = \Omega$, there is some $\rho \in R$ such that $\omega \in \mathrm{ev}(\rho)$. And the existence of this outcome $\omega \in \mathrm{ev}(\rho)$ with $X(\omega) = x$ confirms that $x \in \mathrm{range}(X|\rho)$. So ρ's child $(\rho; X = x)$ is an element of R' whose corresponding event contains ω.

Because any split tree contains only countably many nodes, we know R' is countable. So $\{X = x\}$ can be expressed as a countable union of events in $\{\mathrm{ev}(\sigma) : \sigma \in T\}$. Thus $\{\mathrm{ev}(\sigma) : \sigma \in T\}$ generates all events of the form $\{X = x\}$. □

We are now ready to prove the main theorem of this section: that if we specify probabilities $f(\sigma)$ for all the instantiations in a split tree in a consistent way, then there is a unique probability measure P on $(\Omega, \mathscr{F})$ such that $P(\sigma) = f(\sigma)$ for instantiations in the tree. Perhaps the most obvious approach to proving this theorem would be to impose an arbitrary numbering $X_1, X_2, \ldots$ on the variables, show that the constraints $P(\sigma) = f(\sigma)$ for instantiations in the tree uniquely determine the probabilities of all instantiations of $X_1, \ldots, X_n$ for each n, and then show that these probabilities satisfy the consistency conditions of Kolmogorov's extension theorem. However, that approach turns out to be fairly complicated, and does not provide

much insight.

We take an alternative approach based on the intuition that a split tree represents a program for sampling outcomes. To understand the proof, it helps to imagine a computer that has access to an infinite sequence of random numbers in $(0, 1]$. These numbers govern the choices that the computer makes at each node as it walks along a path from the root. We show that each random number sequence can be associated with an outcome that is consistent with every instantiation on this path. The probability of an event $A \in \mathscr{F}$ is then defined as the probability of the set of random number sequences that yield outcomes in A. Note that this is just a mathematical construction, not an algorithm, because the program runs forever when $\mathcal{V}$ is infinite.

Theorem 3.6. *Let T be a split tree for a set $\mathcal{V}$ of variables that is sufficient for $(\Omega, \mathscr{F})$. Let f be a function from instantiations in T to $[0, 1]$ such that $f(\top) = 1$ and for each non-leaf node σ:*

$$f(\sigma) = \sum_{\rho \in \mathrm{ch}_T(\sigma)} f(\rho) \tag{3.2}$$

Then there is a unique probability measure P on $(\Omega, \mathscr{F})$ such that $P(\sigma) = f(\sigma)$ for every instantiation σ in T.

Proof. First we will construct a probability measure with the desired properties; then we will show it is unique. Let $(U, \mathscr{U}) = \times_{i=1}^{\infty}((0, 1], \mathscr{B}_{(0,1]})$: the space of infinite sequences $u = u_1, u_2, \ldots$ where $u_i \in (0, 1]$. Here $\mathscr{B}_{(0,1]}$ is the Borel σ-field on $(0, 1]$. Let μ be a probability measure on $(U, \mathscr{U})$ such that the coordinate random variables $U_i(u) \triangleq u_i$ are independent and uniformly distributed over $(0, 1]$. Such a probability measure exists by the continuous form of Kolmogorov's extension theorem [Durrett, 1996] (and also by less general results such as Theorem 20.4 of Billingsley [1995]).

For each non-leaf node $\sigma \in T$ with $f(\sigma) > 0$, and each $\rho \in \mathrm{ch}_T(\sigma)$, define:

$$f(\rho|\sigma) \triangleq \frac{f(\rho)}{f(\sigma)} \tag{3.3}$$

Equation 3.2 implies that $\sum_{\rho \in \mathrm{ch}_T(\sigma)} f(\rho|\sigma) = 1$. Now impose an arbitrary ordering $<$ on $\mathrm{ch}_T(\sigma)$, and for each instantiation ρ in that set, define $a(\rho|\sigma) \triangleq \sum_{\rho' < \rho} f(\rho'|\sigma)$ and $b(\rho|\sigma) \triangleq a(\rho|\sigma) + f(\rho|\sigma)$. Note that for any given σ, the intervals $(a(\rho|\sigma), b(\rho|\sigma)]$ are disjoint for distinct children ρ. Also, because the $f(\rho|\sigma)$ values sum to 1 for each σ, the sets $(a(\rho|\sigma), b(\rho|\sigma)]$ for $\rho \in \mathrm{ch}_T(\sigma)$ form a partition of $(0, 1]$.

For non-leaf nodes σ with $f(\sigma) = 0$, we choose a child $\rho^* \in \mathrm{ch}_T(\sigma)$ arbitrarily and let $a(\rho^*|\sigma) = 0$ and $b(\rho^*|\sigma) = 1$. For other children $\rho \in \mathrm{ch}_T(\sigma)$, we let $a(\rho|\sigma)$ and $b(\rho|\sigma)$ both equal 1, defining the empty interval $(1, 1]$.

Now, for each $i \leq |\mathcal{V}|$, we define a function $g_i(u)$ that returns the instantiation we reach at depth i in the split tree when using the random number sequence u:

$$\begin{aligned} g_0(u) &= \top \\ g_{i+1}(u) &= \text{the unique } \rho \in \mathrm{ch}_T(g_i(u)) \\ &\qquad \text{such that } U_{i+1}(u) \in (a(\rho|g_i(u)), b(\rho|g_i(u))] \end{aligned} \tag{3.4}$$

Thus, any random number sequence u yields a non-truncated path $g_0(u), g_1(u), \ldots$. Let $h : (U, \mathscr{U}) \to (\Omega, \mathscr{F})$ be a function such that $h(u)$ is an arbitrary outcome consistent with $\wedge_i g_i(u)$; Lemma 3.3 tells us that at least one such outcome exists. Finally, we define our probability measure P on $(\Omega, \mathscr{F})$ such that $P(A) = \mu(h^{-1}(A))$ for each $A \in \mathscr{F}$.

We must check that the function P defined this way is indeed a probability measure. We will prove this by showing that h is measurable: thus we can regard h as a random variable, and P as the probability measure for this variable induced

by μ (see Section 2.2.3). To show that h is measurable, we must show that the preimage of each element of $\mathscr{F}$ is $\mathscr{U}$-measurable. In fact, because instantiations in T generate $\mathscr{F}$ (Lemma 3.5), it suffices to show that the preimages of the events $\mathrm{ev}(\sigma)$ for $\sigma \in T$ are $\mathscr{U}$-measurable. We will show at the same time that they have the desired probabilities $f(\sigma)$.

We proceed by induction on $|\mathrm{vars}(\sigma)|$. The base case is $\sigma = \top$: then $h^{-1}(\mathrm{ev}(\sigma)) = h^{-1}(\Omega) = U$, since h is a total function on U. So $h^{-1}(\mathrm{ev}(\sigma))$ is measurable and $P(\sigma) = 1 = f(\sigma)$. As the inductive hypothesis, assume that for any n-variable instantiation σ in the tree, $h^{-1}(\mathrm{ev}(\sigma)) \in \mathscr{U}$ and $P(\sigma) = f(\sigma)$. Now let σ be an $(n+1)$-variable instantiation in T. Since σ is in the tree, every non-truncated path either goes through σ or contradicts σ. So if a random variable sequence u yields a non-truncated path consistent with σ, then σ must be the $(n+1)$st node on that path. Thus $h^{-1}(\mathrm{ev}(\sigma)) = \{u : g_{n+1}(u) = \sigma\}$. We also know σ has a parent τ, which is the only n-variable instantiation in T consistent with σ. Thus σ has a unique representation as $(\tau; X = \sigma_X)$ such that τ is in T. By Equation 3.4, we can see that $g_{n+1}(u) = \sigma$ if and only if $g_n(u) = \tau$ and $U_{n+1}(u) \in (a(\sigma|\tau), b(\sigma|\tau)]$. So $h^{-1}(\mathrm{ev}(\sigma)) = h^{-1}(\mathrm{ev}(\tau)) \cap \{u_{n+1} \in (a(\sigma|\tau), b(\sigma|\tau)]\}$. The event $h^{-1}(\mathrm{ev}(\tau))$ is measurable by the inductive hypothesis, and the other event is a half-open interval for u_{n+1}, so $h^{-1}(\sigma)$ is measurable.

Furthermore, the second part of our inductive hypothesis tells us $P(\tau) = f(\tau)$. If $f(\tau) = 0$, then $P(\tau) = 0$, and since σ is an extension of τ, $P(\sigma) = 0$ as well. And by Equation 3.2, $f(\tau) = 0$ also implies that $f(\sigma) = 0$, so $P(\sigma) = f(\sigma)$ in this case. For the case where $f(\tau) > 0$, we exploit the fact that $g_n(u)$ depends only on $u_1, \ldots, u_n$, which are independent of the uniformly distributed u_{n+1}. So $P(\sigma) = P(\tau)(b(\sigma|\tau) - a(\sigma|\tau))$. Now applying the definition of $b(\cdot|\tau)$ and $a(\cdot|\tau)$, we get $P(\sigma) = f(\tau)f(\sigma_X|\tau)$. Equation 3.3 tells us $f(\sigma_X|\tau) = \frac{f(\sigma)}{f(\tau)}$. So we get $P(\sigma) = f(\tau)\frac{f(\sigma)}{f(\tau)} = f(\sigma)$, as desired. Our inductive step is now complete. So we

have shown that h is measurable and that $P(\sigma) = f(\sigma)$ for each σ in the tree.

It just remains to show that this P is the only probability measure on $(\Omega, \mathscr{F})$ that assigns the desired probabilities to instantiations in the tree. By Theorem 2.1, it suffices to show that the set of events $\{\mathrm{ev}(\sigma) : \sigma \in T\}$, with the empty set added, is closed under intersections and generates $\mathscr{F}$. Observe that if two instantiations σ and τ are both on a common path from the root, then $\mathrm{ev}(\sigma) \cap \mathrm{ev}(\tau)$ is either $\mathrm{ev}(\sigma)$ or $\mathrm{ev}(\tau)$, whichever is deeper in the tree. Otherwise, σ and τ are on different paths, so they must disagree on some variable. Thus $\mathrm{ev}(\sigma) \cap \mathrm{ev}(\tau) = \varnothing$. So this set is closed under intersections. The fact that it generates $\mathscr{F}$ is given by Lemma 3.5, so our proof is complete. □

We now have a version of Kolmogorov's extension theorem that allows probabilities to be specified for the instantiations in an arbitrary split tree, rather than requiring probabilities for all finite prefixes of a numbering of $\mathcal{V}$. This theorem also allows the outcome space $(\Omega, \mathscr{F})$ to be arbitrary, as long as $\mathcal{V}$ is sufficient for it. Just as the standard version of Kolmogorov's extension theorem allowed us to derive criteria under which infinite BNs are well-defined, Theorem 3.6 will allow us to provide such criteria for a more general class of models.

3.4 Partition-based models

3.4.1 Definition and examples

Recall our goal of developing declarative probabilistic models for scenarios with contingent dependencies, such as the urn and balls in Example 3.1 or the hurricane preparations in Example 3.2. In these examples, drawing an edge for every dependency that holds under any conditions yields a graph with cycles or infinite ancestor sets. Simply drawing edges between variables does not capture the contingent nature

of the dependency structure.

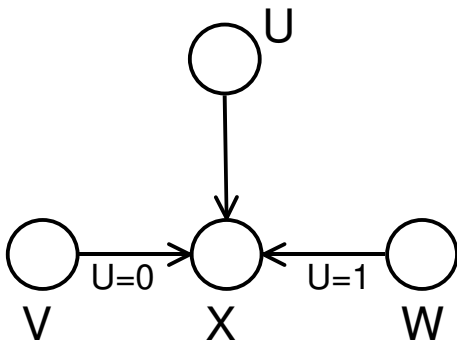


Figure 3.5: A simple BN with contingent dependencies.

To motivate our alternative approach, consider the BN in Fig. 3.5, ignoring for now the labels on the edges. Assuming the variables are binary, the CPD for X can be represented as a table with eight rows, each corresponding to an instantiation of X's three parents. Another way of viewing this is that X's parent set defines a partition of Ω: each CPT row corresponds to a block (i.e., element) of the partition. This may seem like a pedantic rephrasing, but partitions can expose more structure in the CPD. For example, suppose X depends only on V when $U=0$ and only on W when $U=1$. The tabular CPD for X would still be the same size, but now the partition for X only has four blocks: $\{U=0, V=0\}$, $\{U=0, V=1\}$, $\{U=1, W=0\}$, and $\{U=1, W=1\}$. Thus, instead of speaking of a CPD for X given a set of parent variables, we will speak of a CPD for X given a partition of Ω.

Just as we limited ourselves to conditioning on finite sets of discrete variables in Section 2.2.6, we will limit ourselves to countable partitions: that is, partitions with countably many blocks. A partition Λ on a measurable space $(\Omega, \mathscr{F})$ is *measurable* if each block $\lambda \in \Lambda$ is in $\mathscr{F}$.

Definition 3.4. *Let X be a discrete random variable defined on a measurable space $(\Omega, \mathscr{F})$, and let Λ be a countable, measurable partition of Ω. A* conditional probabil-

ity distribution for X given Λ *is a function* $c : \text{range}(X) \times \Lambda \to [0,1]$ *such that for each block* $\lambda \in \Lambda$,

$$\sum_{x \in \text{range}(X)} c(x, \lambda) = 1$$

If P *is a probability measure on* $(\Omega, \mathscr{F})$, *then* c *is a* version of $P(X|\Lambda)$ *if for each* $x \in \text{range}(X)$ *and each* $\lambda \in \Lambda$ *such that* $P(\lambda) > 0$,

$$c(x, \lambda) = P(X = x|\lambda)$$

This definition is consistent with the one given in Definition 2.10 for a CPD given a set of variables. For any set of variables $\mathbf{W}$, $\{\text{ev}(\sigma) : \sigma \in \text{range}(\mathbf{W})\}$ is a partition of Ω; a CPD for X given $\mathbf{W}$ is simply a CPD for X given this partition. Conversely, a CPD for X given a measurable partition Λ is essentially a CPD for X given the random variable V_Λ such that $V_\Lambda(\omega)$ is equal to the unique block in Λ that contains ω.[1]

Building on this generalized definition of a CPD, we introduce a generalization of BNs that we call *partition-based models*.

Definition 3.5. *Let* $\mathcal{V}$ *be a set of discrete random variables that is sufficient for* $(\Omega, \mathscr{F})$. *A* partition-based model Γ *over* $\mathcal{V}$ *specifies, for each* $X \in \mathcal{V}$*:*

- *a countable, measurable partition* Λ_Γ^X *of* Ω, *where we write* $\lambda_\Gamma^X(\omega)$ *to denote the block of the partition that the outcome* ω *belongs to;*
- *a CPD* c_Γ^X *for* X *given* Λ_Γ^X.

Note that any BN $\mathcal{B}$ can be represented as a PBM Γ: for each variable X, we let $\Lambda_\Gamma^X = \{\text{ev}(\sigma) : \sigma \in \text{range}(\text{Pa}_\mathcal{B}(X))\}$. Figure 3.6 describes a PBM Γ for the

[1] The only difference is that the second argument of a CPD given a partition is an event, and the second argument of a CPD given a variable is an instantiation.

urn-and-balls scenario (Example 3.1). Note that we do not have a formal syntax for describing PBMs, so we use a combination of English words, mathematical notation, and tables. The variable N, representing the number of balls, has a prior distribution that does not depend on any other aspects of the outcome. So Λ_Γ^N is the single-block partition $\{\Omega\}$. The distribution for N given this single block is a Poisson distribution with mean 6. Each variable C_i, representing the color of the ith ball, has a partition with two elements: one where (Ball, i) exists, and one where it does not. The BallDrawn variables B_j have an infinite number of partition blocks, one for each possible number of balls. If there are no balls, then $B_j = \mathsf{null}$ with probability one; otherwise, B_j's distribution is uniform over the balls that exist. Finally, each ObsColor variable O_j has three partition blocks, one for each possible value of the term $\mathsf{TrueColor}(\mathsf{BallDrawn}(d))$ (Figure 3.6 uses the notation introduced in Definition 2.7 for the denotation of a term given a model structure and an assignment of values to logical variables). Note that this PBM captures the context-specific dependency structure of this model much better than the graphical model in Figure 3.1 does: each O_j variable has only three partition blocks in the PBM, while it has infinitely many parents in Figure 3.1.

A PBM for the hurricane scenario of Example 3.2 is described in Figure 3.7. The variable F that specifies which city is hit first has the trivial partition $\{\Omega\}$. The preparation variables P_c have four partition blocks: one for the case where c is hit first, and three for the cases where c is not hit first and $\mathsf{Damage}(\mathsf{First})$ takes one of the values $\{\mathsf{Severe}, \mathsf{Mild}, \mathsf{null}\}$. The damage variables D_c have three partition blocks, for the three possible values of $[\mathsf{Prep}]^\omega(c)$. In this PBM, the context-specific structure we are capturing is in the dependency model for the P_c variables: in the BN in Figure 3.2, each P_c variable has one 2-valued parent (F) and one 3-valued parent (a damage variable), but the partition for P_c in the PBM has only four blocks.

For the variable N:

$$\Lambda_\Gamma^N = \{\Omega\}$$
$$c_\Gamma^N(n, \Omega) = \text{Poisson}[6](n)$$

For the variables $\{C_i\}_{i=1}^\infty$:

$$\Lambda_\Gamma^{C_i} = \{\{\omega \in \Omega \,:\, (\mathsf{Ball}, i) \in [\mathsf{Ball}]^\omega\}, \{\omega \in \Omega \,:\, (\mathsf{Ball}, i) \notin [\mathsf{Ball}]^\omega\}\}$$

$c_\Gamma^{C_i}(c,\, \{\omega \in \Omega \,:\, (\mathsf{Ball}, i) \in [\mathsf{Ball}]^\omega\}) =$

	c	
Blue	Green	null
0	0	1

$c_\Gamma^{C_i}(c,\, \{\omega \in \Omega \,:\, (\mathsf{Ball}, i) \notin [\mathsf{Ball}]^\omega\}) =$

	c	
Blue	Green	null
0.5	0.5	0

For the variables $\{B_j\}_{j=1}^k$:

$$\Lambda_\Gamma^{B_j} = \{\{\omega \in \Omega \,:\, |\,[\mathsf{Ball}]^\omega\,| = n\}\}_{n \in \mathbb{N}}$$
$$c_\Gamma^{B_j}(b,\, \{\omega \in \Omega \,:\, |\,[\mathsf{Ball}]^\omega\,| = 0\}) = \begin{cases} 1 & \text{if } b = \mathsf{null} \\ 0 & \text{otherwise} \end{cases}$$
$$c_\Gamma^{B_j}(b,\, \{\omega \in \Omega \,:\, |\,[\mathsf{Ball}]^\omega\,| = n\}) = \begin{cases} 1/n & \text{if } b \in \{(\mathsf{Ball}, 1), \ldots, (\mathsf{Ball}, n)\} \\ 0 & \text{otherwise} \end{cases}, \text{ for } n > 0$$

For the variables $\{O_j\}_{j=1}^k$:

$$\Lambda_\Gamma^{O_j} = \{\{\omega \in \Omega \,:\, [\mathsf{TrueColor}(\mathsf{BallDrawn}(d))]^\omega_{(d \mapsto (\mathsf{Draw}, j))} = c\}\}_{c \in \{\mathsf{Blue}, \mathsf{Green}, \mathsf{null}\}}$$
$$c_\Gamma^{O_j}(o, \{\omega \in \Omega \,:\, [\mathsf{TrueColor}(\mathsf{BallDrawn}(d))]^\omega_{(d \mapsto (\mathsf{Draw}, j))} = c\})$$

$=$

	o		
c	Blue	Green	null
Blue	0.8	0.2	0
Green	0.2	0.8	0
null	0	0	1

Figure 3.6: Partitions and CPDs for each variable in a PBM Γ for the urn-and-balls scenario of Example 3.1, with k draws.

For the variable F:

$$\Lambda_\Gamma^F = \{\Omega\}$$

$c_\Gamma^F(f, \Omega) =$

	f	
A	B	null
0.5	0.5	0

For the variables $\{P_c\}_{c \in \{\text{A,B}\}}$:

$$\Lambda_\Gamma^{P_c} = \{\{\omega \in \Omega : [\text{First}]^\omega = c\}\} \cup \{\{\omega \in \Omega : [\text{First}]^\omega \neq c \text{ and } [\text{Damage}(\text{First})]^\omega = d\}\}_{d \in \{\text{Severe,Mild,null}\}}$$

$c_\Gamma^{P_c}(p, \{\omega \in \Omega : [\text{First}]^\omega = c\}) =$

	p	
High	Low	null
0.5	0.5	0

$c_\Gamma^{P_c}(p, \{\omega \in \Omega : [\text{First}]^\omega \neq c \text{ and } [\text{Damage}(\text{First})]^\omega = d\}) =$

	p		
d	High	Low	null
Severe	0.9	0.1	0
Mild	0.1	0.9	0
null	0	0	1

For the variables $\{D_c\}_{c \in \{\text{A,B}\}}$:

$$\Lambda_\Gamma^{D_c} = \{\{\omega \in \Omega : [\text{Prep}]^\omega (c) = p\}\}_{p \in \{\text{High,Low,null}\}}$$

$c_\Gamma^{D_c}(d, \{\omega \in \Omega : [\text{Prep}]^\omega (c) = p\}) =$

	d		
p	Severe	Mild	null
High	0.2	0.8	0
Low	0.8	0.2	0
null	0	0	1

Figure 3.7: Partitions and CPDs for each variable in a PBM Γ for the hurricane scenario of Example 3.2.

3.4.2 Semantics

As we did with BNs, we would like to specify what it means for a probability measure P on $(\Omega, \mathscr{F})$ to satisfy a PBM Γ. We can easily state that for each variable $X \in \mathcal{V}$, c_Γ^X must be a version of $P(X|\Lambda_\Gamma^X)$. Our other requirement in the BN case (see Definition 2.19) was that P must satisfy the *local Markov property*: each variable is independent of its nondescendants given its parents. But what are the nondescendants of a variable in a PBM? The local Markov property turns out to be not so local, in that it requires having a graph over all the variables so that we can identify the nondescendants of each node.

Since it is not immediately obvious how to generalize a BN's independence properties to PBMs, let us consider the factorization property given by condition (3) in Theorem 2.10. This conditional requires that if $\text{vars}(\sigma)$ is a finite, ancestral set in a BN $\mathcal{B}$, then:

$$P(\sigma) = \prod_{X \in \text{vars}(\sigma)} c_\mathcal{B}^X(\sigma_X, \sigma[\text{Pa}_\mathcal{B}(X)]) \tag{3.5}$$

We would like to state a similar factorization property for PBMs, but this requires some appropriate generalization of the idea of an "ancestral set". Notice that the reason why $\text{vars}(\sigma)$ must be an ancestral set in Equation 3.5 is so that σ assigns values to $\text{Pa}_\mathcal{B}(X)$ for each $X \in \text{vars}(\sigma)$. That is, σ determines which instantiation of $\text{Pa}_\mathcal{B}(X)$ to pass into X's CPD. The analogous requirement in a PBM Γ is that σ must determine an element of the partition Λ_Γ^X. In this case, we say that σ *supports* X.

Definition 3.6. *In a PBM Γ, an instantiation σ* supports *a variable X if σ is achievable and there is some block $\lambda \in \Lambda_\Gamma^X$ such that* $\text{ev}(\sigma) \subseteq \lambda$.

If σ supports X, then we will write $\lambda_\Gamma^X(\sigma)$ for the unique element of Λ_Γ^X that has $\text{ev}(\sigma)$ as a subset. Intuitively, σ supports X if knowing σ is enough to tell us which

block of Λ_Γ^X we're in; then we can use X's CPD to obtain a conditional probability for each possible value of X. In Example 3.1, $(B_1 = 8, C_8 = \mathsf{Blue})$ supports O_1, but $(B_1 = 4, C_8 = \mathsf{Blue})$ does not. In an ordinary BN, any instantiation of the parents of X supports X.

Note that if σ supports X and ρ is an extension of σ, then ρ supports X as well — unless ρ is not achievable. We require σ to be achievable in Definition 3.6 because the event corresponding to an unachievable instantiation — namely the empty set — is a subset of *every* element of Λ_Γ^X, and thus does not uniquely identify a partition block.

So an instantiation that supports X in a PBM is analogous to an instantiation of X's parents in a BN. We can now define a notion analogous to that of an ancestral set.

Definition 3.7. *In a PBM* Γ*, an instantiation* σ *is* self-supporting *if for each* $X \in \mathrm{vars}(\sigma)$*,* σ *supports* X*.*

If a set of variables is an ancestral set in a BN, then any instantiation of those variables is self-supporting in the PBM representation of that BN. Note that being self-supporting is a property of instantiations, not of sets of variables. Some instantiations of a given set of variables may be self-supporting while others are not: for instance, in Figure 3.5, the instantiation $(N = 9,\ B_1 = 8,\ C_8 = \mathsf{Blue},\ O_1 = \mathsf{Blue})$ is self-supporting, but $(N = 9,\ B_1 = 4,\ C_8 = \mathsf{Blue},\ O_1 = \mathsf{Blue})$ is not. The need to talk about instantiations rather than variables is a consequence of our choice to represent contingent dependencies.

We now have a partition-based analogue of the notion of an ancestral set. It is not immediately obvious how this helps us to generalize the local Markov property of BNs, which talks about the nondescendants of a variable. However, it turns out that the local Markov property can be reformulated in terms of ancestral sets.

The property states that under any probability measure P that satisfies a BN $\mathcal{B}$, $X \perp\!\!\!\perp_P \text{NonDesc}_{\mathcal{B}}(X) \,|\, \text{Pa}_{\mathcal{B}}(X)$. This is equivalent to saying that for every finite subset $\mathbf{W} \subseteq \text{NonDesc}_{\mathcal{B}}(X)$, $X \perp\!\!\!\perp_P \mathbf{W} \,|\, \text{Pa}_{\mathcal{B}}(X)$. But in a BN where every node has finitely many ancestors, each finite $\mathbf{W} \subseteq \text{NonDesc}_{\mathcal{B}}(X)$ can be extended to a finite ancestral set $\mathbf{W}' \subseteq \text{NonDesc}_{\mathcal{B}}(X)$. The ancestral sets that are subsets of $\text{NonDesc}_{\mathcal{B}}(X)$ are precisely those ancestral sets that do not contain X. So for BNs that have topological numberings, the local Markov property is equivalent to the following: if $\mathbf{W}$ is a finite ancestral set that does not contain X, then $X \perp\!\!\!\perp_P \mathbf{W} \,|\, \text{Pa}_{\mathcal{B}}(X)$. This observation suggests the following definition for the semantics of a PBM.

Definition 3.8. *Let Γ be a PBM for a set of variables $\mathcal{V}$ defined on an outcome space $(\Omega, \mathscr{F})$. A probability measure P on $(\Omega, \mathscr{F})$* satisfies Γ *if, for each $X \in \mathcal{V}$,*

i. c_Γ^X is a version of $P(X|\Lambda_\Gamma^X)$;

ii. for every finite, self-supporting instantiation σ that does not instantiate X, $X \perp\!\!\!\perp_P \sigma \,|\, \Lambda_\Gamma^X$.

We will refer to part (ii) of this definition as the local Markov property for PBMs. Note that it does not specify independence between variables, but rather between a variable and certain events. A PBM is *well-defined* if there is exactly one probability measure that satisfies it. The rest of this section is devoted to identifying conditions under which a PBM is guaranteed to be well-defined.

3.4.3 Supportive split trees

Recall that most of our results about BNs rely on the assumption that the BN has a topological numbering. To prove further results about PBMs, we will need an analogous assumption. We already have a contingent analogue of a numbering,

namely the split tree, introduced in Section 3.3. We would like to define a class of split trees that are analogous to topological numberings.

In a topological numbering for a BN $\mathcal{B}$, the predecessor set for any variable X includes $\mathrm{Pa}_{\mathcal{B}}(X)$. One way to generalize this condition to split trees is to require that if a node σ splits on a variable X, then σ must support X. This is essentially the condition we will use, but it turns out that we can weaken it slightly (and this weakening is useful; see Appendix 3.B). Specifically, we will not require σ to support its split variable if it is *directly disallowed*, in the following sense.

Definition 3.9. *An instantiation σ is* directly disallowed *in a PBM Γ if there is some $X \in \mathrm{vars}(\sigma)$ and some sub-instantiation τ of σ that supports X, such that $c^X_\Gamma(\sigma_X, \lambda^X_\Gamma(\tau)) = 0$.*

If σ is directly disallowed in Γ, then any probability measure that satisfies Γ assigns probability zero to σ. To see this, observe that if c^X_Γ is a version of $P(X|\Lambda^X_\Gamma)$ and $c^X_\Gamma(\sigma_X, \lambda^X_\Gamma(\tau)) = 0$, then $P(\lambda^X_\Gamma(\tau) \cap \{X = \sigma_X\}) = 0$. Since $\mathrm{ev}(\tau) \subseteq \lambda^X_\Gamma(\tau)$, it follows that $P(\tau; X = x) = 0$, and thus $P(\sigma) = 0$ if $(\tau; X = x)$ is a sub-instantiation of σ.

Because any directly disallowed instantiation σ has probability zero, the PBM determines the value of $P(\sigma; X = x)$ even when σ does not support X: $P(\sigma; X = x) = 0$. So we do not need to require that directly disallowed nodes support their split variables. We define a contingent analogue of a topological numbering as follows:

Definition 3.10. *A* supportive split tree *for a PBM Γ is a split tree for Γ's variables in which each non-leaf instantiation σ either supports its split variable X^σ_T, or is directly disallowed.*

3.4.4 Supportive numberings

In Section 2.4.2, our first step in deriving well-definedness conditions for BNs was to prove that a BN determines the probabilities of the instantiations of finite prefixes of any topological numbering. We can prove a similar result for the instantiations in a supportive split tree for a PBM. This result is based on the fact that if an instantiation in a supportive split tree is not directly disallowed, then it has a *supportive numbering*.

Definition 3.11. *A* supportive numbering *for an instantiation σ in a PBM is a numbering π of* $\text{vars}(\sigma)$ *such that for each $X \in \text{vars}(\sigma)$, the instantiation $\sigma[\text{Pred}_\pi[X]]$ supports X.*

If a path from the root of a supportive split tree contains no directly disallowed nodes, then the order in which variables are instantiated serves as a supportive numbering for instantiations on that path. This ordering also serves as a supportive numbering for the first directly disallowed instantiation on a path. Subsequent instantiations on the same path may not have supportive numberings, but they are all directly disallowed. Thus, any node in a supportive split tree either has a supportive numbering or is directly disallowed.

Having a supportive numbering is closely related to being self-supporting — in fact, in our IJCAI paper on BLOG [Milch *et al.*, 2005a], we used the term "self-supporting" to refer to instantiations that have a supportive numbering. Unfortunately, the two notions are not equivalent. It is true that if an instantiation has a supportive numbering and is achievable, then it is self-supporting (unachievable instantiations cannot be self-supporting because they do not support any variables). But the converse does not hold: an achievable instantiation that is self-supporting may fail to have a supportive numbering. As an example, consider a PBM Γ with just one random variable, X, such that $\Lambda_\Gamma^X = \{\{X = 0\}, \{X = 1\}\}$. That is, the condi-

tional distribution for X depends on X itself. A PBM such as this one will generally not be well-defined, but it is still a PBM. Here the instantiation $\sigma \triangleq (X = 1)$ is self-supporting, but it does not have a supportive numbering. For any numbering π, $\sigma[\text{Pred}_\pi[X]]$ is just $\top$, which does not support X in this PBM.

We will now prove a factorization property for instantiations that have supportive numberings.

Lemma 3.7. *Let P be a probability measure that satisfies a PBM Γ for a set of variables $\mathcal{V}$. If σ is a finite, achievable instantiation on $\mathcal{V}$ that has a supportive numbering, then:*

$$P(\sigma) = \prod_{X \in \text{vars}(\sigma)} c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\sigma)) \tag{3.6}$$

Proof. We proceed by induction on $|\text{vars}(\sigma)|$. The base case, where $\text{vars}(\sigma) = \varnothing$, is trivial: we get $P(\top) = 1$. Now suppose the lemma holds for all instantiations of size n, and consider any achievable instantiation σ of size $n+1$ that has a supportive numbering π. Let Y be the last element of $\text{vars}(\sigma)$ according to π.

If $P(\sigma_{-Y}) = 0$, then by the inductive hypothesis, there is some $X \in (\text{vars}(\sigma) \setminus Y)$ such that $c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\sigma_{-Y})) = 0$. Because σ is achievable, $\lambda_\Gamma^X(\sigma) = \lambda_\Gamma^X(\sigma_{-Y})$. Thus the right hand side of Equation 3.6 evaluates to zero for σ as well. And since $\text{ev}(\sigma) \subseteq \text{ev}(\sigma_{-Y})$, we know $P(\sigma) = 0$ too, so Equation 3.6 is satisfied in this case.

Otherwise, by the definition of conditional probability, $P(\sigma) = P(Y = \sigma_Y | \sigma_{-Y}) P(\sigma_{-Y})$. Now since π is supportive and $\sigma_{-Y} = \sigma[\text{Pred}_\pi[Y]]$, we know σ_{-Y} supports Y. Since $\text{ev}(\sigma_{-Y}) \subseteq \lambda_\Gamma^Y(\sigma_{-Y})$, we can express σ_{-Y} as $\text{ev}(\sigma_{-Y}) \cap \lambda_\Gamma^Y(\sigma_{-Y})$. Therefore:

$$P(\sigma) = P(Y = \sigma_Y \mid \text{ev}(\sigma_{-Y}) \cap \lambda_\Gamma^Y(\sigma_{-Y}))\, P(\sigma_{-Y})$$

We know that σ_{-Y} has a supportive numbering, and it must be achievable because σ is achievable. So σ_{-Y} is a finite, self-supporting instantiation that does not instan-

tiate Y. So by the local Markov property for PBMs, $Y \perp\!\!\!\perp_P \sigma_{-Y} \mid \Lambda^Y_\Gamma$. Thus:

$$P(\sigma) = P(Y = \sigma_Y \mid \lambda^Y_\Gamma(\sigma_{-Y}))\, P(\sigma_{-Y})$$

Then by the inductive hypothesis,

$$P(\sigma) = P(Y = \sigma_Y \mid \lambda^Y_\Gamma(\sigma_{-Y})) \prod_{X \in \text{vars}(\sigma_{-Y})} c^X_\Gamma(\sigma_X, \lambda^X_\Gamma(\sigma_{-Y}))$$

Finally, because c^Y_Γ is a version of $P(Y|\Lambda^Y_\Gamma)$, we have:

$$P(\sigma) = \prod_{X \in \text{vars}(\sigma)} c^X_\Gamma(\sigma_X, \lambda^X_\Gamma(\sigma_{-Y}))$$

Since $\lambda^X_\Gamma(\sigma) = \lambda^X_\Gamma(\sigma_{-Y})$ for all variables X that σ_{-Y} supports, we have shown that $P(\sigma)$ satisfies Equation 3.6. □

3.4.5 Criteria for satisfying a PBM

We now know that if T is a supportive split tree for a PBM Γ, then Γ fully determines the probabilities of all instantiations in the tree. By applying Theorem 3.6, we will be able to show that there is a unique probability measure on Γ's outcome space that assigns the specified probabilities to all the instantiations. However, we must also show that any such probability measure satisfies Γ. We faced the same issue with BNs, where we had to show that assigning the correct probabilities to all instantiations of prefixes of a topological numbering was sufficient for satisfying a BN. As in the BN case, we will prove the equivalence of three conditions on a probability measure P: P satisfies a given PBM; P assigns the correct probabilities to instantiations in a supportive split tree; and P satisfies a certain factorization property for all finite, self-supporting instantiations.

Lemma 3.8. *Let Γ be a PBM for a set of variables $\mathcal{V}$ defined on an outcome space $(\Omega, \mathscr{F})$, let P be a probability measure on $(\Omega, \mathscr{F})$, and let T be a supportive split tree for Γ. Suppose that for every instantiation σ in T that has a supportive numbering, $P(\sigma)$ satisfies Equation 3.6. Then Equation 3.6 holds for every finite, self-supporting instantiation σ on $\mathcal{V}$.*

Proof. Our proof proceeds by induction, and actually involves proving a stronger statement than the one given in the lemma: if τ is a node in T, and σ is any finite instantiation such that $\text{vars}(\sigma) \cap \text{vars}(\tau) = \varnothing$ and $(\tau; \sigma)$ is self-supporting, then:

$$P(\tau; \sigma) = P(\tau) \prod_{X \in \text{vars}(\sigma)} c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\tau; \sigma)) \tag{3.7}$$

Note that if τ is the empty instantiation (the root of T), then $P(\tau) = 1$ and we get Equation 3.6.

Our induction is on the size of $\text{vars}(\sigma)$. If $|\text{vars}(\sigma)| = 0$, then Equation 3.7 is just $P(\tau) = P(\tau)$, which is trivially true. Now suppose Equation 3.7 holds for instantiations of size n, and consider any σ of size $n+1$. Let τ be a node in T such that $\text{vars}(\sigma) \cap \text{vars}(\tau) = \varnothing$ (which implies that the path to τ does not split on any variables in $\text{vars}(\sigma)$) and $(\tau; \sigma)$ is self-supporting. Let R be the set of nodes in the sub-tree induced by τ that are the first along their path from τ to split on a variable in $\text{vars}(\sigma)$. Because every non-truncated path splits on every variable, there is an element of R on every non-truncated path from τ, and thus (by Lemma 3.4), $\{\text{ev}(\rho) : \rho \in R\}$ is a partition of $\text{ev}(\tau)$.

Now consider any node $\rho \in R$ that is not directly disallowed, and let $W = X_T^\rho$. Let ρ' be the child of ρ such that $\rho'_W = \sigma_W$. Then $(\rho; \sigma) = (\rho'; \sigma_{-W})$. Also, $(\rho; \sigma)$ is self-supporting: we know this because ρ is self-supporting, and every variable in $\text{vars}(\sigma)$ is supported by $(\tau; \sigma)$, which is a sub-instantiation of $(\rho; \sigma)$. So, applying

the inductive hypothesis to σ_{-W}, we have:

$$P(\rho;\sigma) = P(\rho') \prod_{X \in (\mathrm{vars}(\sigma) \setminus W)} c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\rho;\sigma))$$

Because ρ is not directly disallowed, we know it has a supportive numbering. And since T is supportive, we know ρ supports W, so this numbering can be extended to a supportive numbering of ρ'. So P satisfies Equation 3.6 for both ρ and ρ'. This implies $P(\rho') = P(\rho)c_\Gamma^W(\rho'_W, \lambda_\Gamma^W(\rho'))$. We chose ρ' such that $\rho'_W = \sigma_W$; also, since ρ supports W, we have $\lambda_\Gamma^W(\rho') = \lambda_\Gamma^W(\rho)$. So we can replace $P(\rho')$ with $P(\rho)c_\Gamma^W(\sigma_W, \lambda_\Gamma^W(\rho))$ in our previous equation, yielding:

$$P(\rho;\sigma) = P(\rho)c_\Gamma^W(\sigma_W, \lambda_\Gamma^W(\rho)) \prod_{X \in (\mathrm{vars}(\sigma) \setminus W)} c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\rho;\sigma))$$

Note that we have both $\mathrm{ev}(\rho;\sigma) \subseteq \mathrm{ev}(\rho) \subseteq \lambda_\Gamma^W(\rho)$ and $\mathrm{ev}(\rho;\sigma) \subseteq \mathrm{ev}(\tau;\sigma) \subseteq \lambda_\Gamma^W(\tau;\sigma)$. So because the elements of Λ_Γ^W are disjoint (and $(\rho;\sigma)$ is self-supporting and therefore achievable), it must be that $\lambda_\Gamma^W(\rho) = \lambda_\Gamma^W(\tau;\sigma)$. Similarly, for each $X \in \mathrm{vars}(\sigma) \setminus W$, $\lambda_\Gamma^X(\rho;\sigma) = \lambda_\Gamma^X(\tau;\sigma)$. Thus:

$$P(\rho;\sigma) = P(\rho) \prod_{X \in \mathrm{vars}(\sigma)} c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\tau;\sigma))$$

This equation also holds trivially for nodes ρ that are directly disallowed: $P(\rho)$ must be zero because the first directly disallowed instantiation on the path to ρ has zero probability according to Equation 3.6, and then $P(\rho;\sigma)$ must be zero as well. Since the equation holds for every $\rho \in R$, and $\{\mathrm{ev}(\rho) : \rho \in R\}$ is a partition of $\mathrm{ev}(\tau)$, we

have:

$$\begin{aligned} P(\tau;\sigma) &= \sum_{\rho \in R} P(\rho;\sigma) \\ &= \sum_{\rho \in R} P(\rho) \prod_{X \in \text{vars}(\sigma)} c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\tau;\sigma)) \\ &= \left(\sum_{\rho \in R} P(\rho)\right) \prod_{X \in \text{vars}(\sigma)} c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\tau;\sigma)) \\ &= P(\tau) \prod_{X \in \text{vars}(\sigma)} c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\tau;\sigma)) \end{aligned}$$

This is the desired statement Equation 3.7. □

Given that $P(\sigma)$ satisfies Equation 3.6 for every finite, self-supporting instantiation σ, it is trivial to show that $P(X = x|\sigma) = c_\Gamma^X(x, \lambda_\Gamma^X(\sigma))$ whenever σ is a finite, self-supporting instantiation that supports X. However, it takes a bit of work to show that P satisfies the PBM. The CPD constraint of Definition 3.8 involves a partition block rather than an instantiation σ, and the local Markov property involves a finite, self-supporting instantiation that may or may not support X.

Lemma 3.9. *Let Γ be a PBM for a set of variables $\mathcal{V}$ defined on an outcome space $(\Omega, \mathscr{F})$. Assume Γ has a supportive split tree. If P is a probability measure on $(\Omega, \mathscr{F})$ such that $P(\sigma)$ satisfies Equation 3.6 for every finite, self-supporting instantiation σ on $\mathcal{V}$, then P satisfies Γ.*

Proof. Let T be a supportive split tree for Γ, and consider any variable $X \in \mathcal{V}$. Let x be any value in range (X), λ be any partition block in Λ_Γ^X such that $P(\lambda) > 0$, and σ be any finite, self-supporting instantiation that does not instantiate X. We will show that $P(\sigma; X = x \mid \lambda) = c_\Gamma^X(x, \lambda) P(\sigma|\lambda)$. First, let R be the set of instantiations in T that split on X. There is one element of R on every non-truncated path from the root of T, so by Lemma 3.4, $\{\text{ev}(\rho) : \rho \in R\}$ is a countable partition of Ω.

Therefore:

$$\begin{aligned} P(\sigma; X = x|\lambda) &= \sum_{\rho \in R} P(\sigma; \rho; X = x|\lambda) \\ &= \frac{1}{P(\lambda)} \sum_{\rho \in R} P(\text{ev}\,(\sigma; \rho; X = x) \cap \lambda) \end{aligned}$$

If an instantiation $\rho \in R$ is directly disallowed, $P(\rho)$ must be zero. To see this, note that the first directly disallowed instantiation on the path to ρ is self-supporting; and if a self-supporting instantiation is directly disallowed, then it must have probability zero under Equation 3.6. So we can remove directly disallowed instantiations from R without changing the sum in the equation above. Now consider an instantiation $\rho \in R$ that is not directly disallowed. By the definition of a supportive split tree, ρ must support X. So either $\text{ev}\,(\rho) \subseteq \lambda$ or $\text{ev}\,(\rho) \cap \lambda = \varnothing$. For those instantiations ρ with $\text{ev}\,(\rho) \cap \lambda = \varnothing$, we can see that $P(\text{ev}\,(\sigma; \rho; X = x) \cap \lambda) = 0$. Finally, this probability is also zero if ρ contradicts σ. So without changing the result of the summation, we can replace R with the set R' of instantiations in R that represent subsets of λ, are consistent with σ, and are not directly disallowed.

$$P(\sigma; X = x|\lambda) = \frac{1}{P(\lambda)} \sum_{\rho \in R'} P(\sigma; \rho; X = x)$$

Now note that for each $\rho \in R'$, both $(\sigma; \rho)$ and $(\sigma; \rho; X = x)$ are self-supporting instantiations. So by Equation 3.6,

$$\begin{aligned} P(\sigma; X = x|\lambda) &= \frac{1}{P(\lambda)} \sum_{\rho \in R'} c^X_\Gamma(x, \lambda^X_\Gamma(\rho)) P(\sigma; \rho) \\ &= \frac{1}{P(\lambda)}\, c^X_\Gamma(x, \lambda) \sum_{\rho \in R'} P(\sigma; \rho) \end{aligned}$$

Here we have used the fact that each $\rho \in R'$ represents a subset of λ. For the

same reason, we can write the sum in the previous equation more evocatively as $\sum_{(\rho \in R')} P(\mathrm{ev}\,(\sigma) \cap \lambda \cap \mathrm{ev}\,(\rho))$. It then becomes clear, given our construction of R', that this sum is equal to $P(\mathrm{ev}\,(\sigma) \cap \lambda)$. So we have:

$$\begin{aligned} P(\sigma; X = x|\lambda) &= \frac{1}{P(\lambda)}\, c^X_\Gamma(x,\lambda) P(\mathrm{ev}\,(\sigma) \cap \lambda) \\ &= c^X_\Gamma(x,\lambda) P(\sigma|\lambda) \end{aligned}$$

This is the equation we set out to prove. Note that for any given $x \in \mathrm{range}\,(X)$ and $\lambda \in \Lambda^X_\Gamma$, setting $\sigma = \top$ (which is trivially a self-supporting instantiation) yields $P(X = x|\lambda) = c^X_\Gamma(x,\lambda)$. Therefore c^X_Γ is a version of $P(X|\Lambda^X_\Gamma)$. Applying this conclusion to the equation we derived yields:

$$P(\sigma; X = x \mid \lambda) \;=\; P(X = x \mid \lambda)\, P(\sigma \mid \lambda)$$

Therefore $X \perp\!\!\!\perp_P \sigma \mid \Lambda^X_\Gamma$. So both conditions of Definition 3.8 are satisfied, and P satisfies Γ. □

With these lemmas in hand, we can now prove the following equivalence result.

Theorem 3.10. *Let Γ be a PBM for a set of random variables $\mathcal{V}$ defined on an outcome space $(\Omega, \mathscr{F})$, and let P be a probability measure on $(\Omega, \mathscr{F})$. If T is a supportive split tree for Γ, then the following three criteria are equivalent:*

1. *P satisfies Γ (in the sense of Definition 3.8).*
2. *For every instantiation σ in the tree T that has a supportive numbering, $P(\sigma)$ satisfies Equation 3.6.*
3. *For every finite, self-supporting instantiation σ on $\mathcal{V}$, $P(\sigma)$ satisfies Equation 3.6.*

Proof. We use a circle of implications.

(1) ⇒ (2): If P satisfies Γ, then Lemma 3.7 implies that $P(\sigma)$ satisfies Equation 3.6 for every finite, achievable instantiation σ that has a supportive numbering. Every instantiation in a split tree is finite and (by Lemma 3.2) achievable. So in particular, $P(\sigma)$ satisfies Equation 3.6 for every instantiation in T that has a supportive numbering.

(2) ⇒ (3): This is Lemma 3.8.

(3) ⇒ (1): This is Lemma 3.9. □

We have a reason to be interested in each of the criteria in Theorem 3.10. Criterion (1) is the definition of PBM semantics based on CPDs and a local Markov property, analogous to the definition for BNs in Section 2.4. Criterion (2) gives us a connection to Theorem 3.6, which says that any consistent assignment of probabilities to the instantiations in a split tree defines a unique probability measure. And criterion (3) is used in the AI/Stats paper where we introduced PBMs [Milch *et al.*, 2005b] as the definition for when a probability measure satisfies a PBM. When we wrote that paper, we had not yet figured out a straightforward way of stating a local Markov property for PBMs. Theorem 3.10 confirms that the definitions are equivalent when the PBM has a supportive split tree.

3.4.6 CPDs and outcome spaces

We have now shown that if a PBM Γ has a supportive split tree T, then a probability measure P satisfies Γ if and only if $P(\sigma)$ satisfies Equation 3.6 for every instantiation σ in T that has a supportive numbering. But before we can apply our results about split trees to show that Γ is well-defined, we must check that the probabilities assigned to instantiations in T by Equation 3.6 are consistent. For any PBM Γ, we can define a function f_Γ on achievable instantiations that have a supportive

numbering or are directly disallowed, as follows:

$$f_\Gamma(\sigma) = \begin{cases} \prod_{X \in \text{vars}(\sigma)} c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\sigma)) & \text{if } \sigma \text{ has a supportive numbering} \\ 0 & \text{if } \sigma \text{ is directly disallowed} \end{cases} \tag{3.8}$$

If σ has a supportive numbering and is directly disallowed, then the two parts of this definition agree. To see this, suppose σ is directly disallowed: that is, it has a sub-instantiation τ and a variable $X \in \text{vars}(\sigma)$ such that $c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\tau)) = 0$. Since σ is an achievable extension of τ, we know $\lambda_\Gamma^X(\sigma) = \lambda_\Gamma^X(\tau)$, so the factor for X in the product expression for σ is zero as well.

If P satisfies a PBM Γ and T is a supportive split tree for Γ, then Theorem 3.10 implies that $P(\sigma) = f_\Gamma(\sigma)$ for every instantiation σ in T. So we must check that this function f_Γ satisfies the consistency condition of Theorem 3.6. That is, for each non-leaf node σ, we must show that:

$$\sum_{\rho \in \text{ch}_T(\sigma)} f_\Gamma(\rho) = f_\Gamma(\sigma)$$

Suppose σ is an instantiation in T that is not directly disallowed and that splits on a variable X. Using Equation 3.8 and the fact that σ has one child for each element of range $(X|\sigma)$, it is easy to show that:

$$\sum_{\rho \in \text{ch}_T(X)} f_\Gamma(\rho) = f_\Gamma(\sigma) \sum_{x \in \text{range}(X|\sigma)} c_\Gamma^X(x, \lambda_\Gamma^X(\sigma))$$

However, nothing we have said so far implies that $c_\Gamma^X(x, \lambda_\Gamma^X(\sigma))$ sums to one over range $(X|\sigma)$. The definition of a CPD just implies that $c_\Gamma^X(x, \lambda_\Gamma^X(\sigma))$ sums to one over range (X), but some values of x that are not in range $(X|\sigma)$ may get positive probability. Thus, we need an additional assumption to ensure that f_Γ does not

assign positive probability to unachievable instantiations.

One plausible assumption is that for each $\lambda \in \Lambda_\Gamma^X$, $c_\Gamma^X(x, \lambda)$ sums to one over range $(X|\lambda)$. However, this turns out to be insufficient. The problem is that a PBM's partitions may fail to respect the structure of the outcome space. For example, suppose we have two binary variables X and Y, and our outcome space Ω consists of just those instantiations σ of $\{X, Y\}$ such that $\sigma_X \neq \sigma_Y$. Now suppose we define a PBM Γ in which neither variable depends on the other: each variable has the trivial partition $\{\Omega\}$. The assumption that $c_\Gamma^X(x, \Omega)$ sums to one over range $(X|\Omega)$ is vacuous, because range $(X|\Omega) = \{0, 1\}$; the same thing happens for $c_\Gamma^Y(y, \Omega)$. So the unachievable instantiations $(X = 1, Y = 1)$ and $(X = 0, Y = 0)$ may end up with positive probability. We need a stronger assumption to rule out this phenomenon. It turns out that the following condition does the trick:

Definition 3.12. *A PBM* Γ respects its outcome space *if, whenever* π *is a supportive numbering of an instantiation* σ *and* σ *is unachievable, then* π yields a zero factor *on* σ *under* Γ*: that is,* $c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\sigma[\mathrm{Pred}_\pi[X]])) = 0$ *for some* $X \in \mathrm{vars}(\sigma)$.

Note that if some supportive numbering of σ yields a zero factor, then σ is directly disallowed. However, there are counterintuitive examples where the converse does not hold.[2]

Lemma 3.11. *Let* Γ *be a PBM that respects its outcome space, and let* T *be a supportive split tree for* Γ*. Then for every non-leaf instantiation* σ *in* T*:*

$$f_\Gamma(\sigma) = \sum_{\rho \in \mathrm{ch}_T(\sigma)} f_\Gamma(\rho) \tag{3.9}$$

[2]For example, suppose the instantiation $\sigma \triangleq (X = 1, Y = 1)$ is unachievable. Because σ is not achievable, its sub-instantiations may yield different blocks in Y's partition. Specifically, it is possible that both $(X = 1)$ and $(Y = 1)$ support Y, but $\lambda_\Gamma^Y(X = 1) = \lambda_1$ with $c_\Gamma^Y(1, \lambda_1) > 0$, while $\lambda_\Gamma^Y(Y = 1) = \lambda_2$ with $c_\Gamma^Y(1, \lambda_2) = 0$. Then a supportive numbering that puts X before Y may not yield a zero factor, even though σ is directly disallowed.

Proof. Let σ be a non-leaf instantiation in T, and let X be the variable that σ splits on. Because T is self-supporting, we know that either σ is directly disallowed, or σ has a supportive numbering and supports X. If σ is directly disallowed, then all its children are directly disallowed as well. So by Equation 3.8, $f_\Gamma(\sigma) = 0$, and also $f_\Gamma(\rho) = 0$ for each $\rho \in \mathrm{ch}_T(\sigma)$. So both sides of Equation 3.9 are zero.

Now suppose σ is not directly disallowed. Because $c_\Gamma^X(x, \lambda)$ sums to one over $\mathrm{range}(X)$ (which is countable), we can write:

$$f_\Gamma(\sigma) = f_\Gamma(\sigma) \sum_{x \in \mathrm{range}(X)} c_\Gamma^X(x, \lambda_\Gamma^X(\sigma))$$

Consider any $x \in \mathrm{range}(X) \setminus \mathrm{range}(X|\sigma)$. By the definition of $\mathrm{range}(X|\sigma)$, we know $(\sigma; X = x)$ is not achievable. Because σ is a node in a supportive split tree and is not directly disallowed, we know it has a supportive numbering, say π. Putting X at the end of this numbering yields a supportive numbering π' for $(\sigma; X = x)$. Because Γ respects its outcome space, it follows that π' yields a zero factor. This factor cannot involve the CPD for any variable in $\mathrm{vars}(\sigma)$, because then π would also yield a zero factor and σ would be directly disallowed. So the zero factor must be $c_\Gamma^X(x, \lambda_\Gamma^X(\sigma))$. Thus, we can remove all x values in $\mathrm{range}(X) \setminus \mathrm{range}(X|\sigma)$ from the summation in our last equation without changing the result. This yields:

$$\begin{aligned}
f_\Gamma(\sigma) &= f_\Gamma(\sigma) \sum_{x \in \mathrm{range}(X|\sigma)} c_\Gamma^X(x, \lambda_\Gamma^X(\sigma)) \\
&= \sum_{x \in \mathrm{range}(X|\sigma)} f_\Gamma(\sigma) c_\Gamma^X(x, \lambda_\Gamma^X(\sigma)) \\
&= \sum_{x \in \mathrm{range}(X|\sigma)} f_\Gamma(\sigma; X = x) \quad \text{by Equation 3.8}
\end{aligned}$$

Finally, recalling that in any split tree, $\mathrm{ch}_T(\sigma) = \{(\sigma; X = x) : x \in \mathrm{range}(X|\sigma)\}$, we get Equation 3.9. □

3.4.7 Sufficient condition for being well-defined

Given all these results, it is now straightforward to prove our main theorem about PBMs.

Theorem 3.12. *Let Γ be a PBM that respects its outcome space. If there exists a supportive split tree for Γ, then Γ is well-defined.*

Proof. Let T be a supportive split tree for T. The function f_Γ defined in Equation 3.8 assigns a number between 0 and 1 to each instantiation in this tree. Furthermore, Lemma 3.11 ensures that f_Γ satisfies the conditions of Theorem 3.6 on this tree. So by Theorem 3.6, there is a unique probability measure P on the outcome space of Γ such that $P(\sigma) = f_\Gamma(\sigma)$ for all σ in T. By the definition of f_Γ, this P satisfies Equation 3.6 for each instantiation in T that has a supportive numbering. So by Theorem 3.10, P satisfies Γ.

To show uniqueness, consider any probability measure P' that satisfies Γ. By Theorem 3.10, P' must satisfy Equation 3.6 for each instantiation in T that has a supportive numbering. And as we argued after the definition of a directly disallowed instantiation, any probability measure that satisfies Γ must assign probability zero to directly disallowed instantiations. So $P'(\sigma) = f_\Gamma(\sigma)$ for all σ in T. Thus the uniqueness part of Theorem 3.6 implies $P' = P$. So there is exactly one probability measure that satisfies Γ. □

3.4.8 Supportive numberings for outcomes

It may not always be obvious when a PBM has a supportive split tree. However, it may be easier to show that each outcome of a PBM has a supportive numbering. We say that an outcome $\omega \in \Omega$ has a supportive numbering if the complete instantiation corresponding to ω has a supportive numbering. It turns out that if the PBM

respects its outcome space, then supportive numberings for individual outcomes can always be used to construct a supportive split tree — which, by definition, covers all outcomes.

Theorem 3.13. *Let Γ be a PBM over a set of random variables $\mathcal{V}$ defined on an outcome space $(\Omega, \mathscr{F})$. If Γ respects its outcome space and every outcome in Ω has a supportive numbering, then Γ has a supportive split tree and hence is well-defined.*

Proof. Let π_g be an arbitrary "global" numbering of $\mathcal{V}$. We will construct a supportive split tree T for Γ inductively. Let the root node of the tree be the empty instantiation. Then, for each incomplete instantiation σ in the tree, define the children of σ as follows. If σ is directly disallowed, then let X_T^σ be the first variable in the numbering π_g that is not in $\text{vars}(\sigma)$. Otherwise, let X_T^σ be the first variable not in $\text{vars}(\sigma)$ that is supported by σ. Let the children of σ be $\{(\sigma; X_T^\sigma = x) : x \in \text{range}(X_T^\sigma|\sigma)\}$.

To show that this construction is well-defined, we must show that for every incomplete instantiation σ that is not directly disallowed, there is some variable not in $\text{vars}(\sigma)$ that is supported by σ. To see this, assume for contradiction that such an instantiation σ does not support any variable not in $\text{vars}(\sigma)$. Our construction guarantees that each instantiation in the tree is achievable, by the same argument as in the proof of Lemma 3.2. So let ω be any outcome in Ω that is consistent with σ. By hypothesis, ω has a supportive numbering π_ω. Because σ is incomplete, there is some variable not in $\text{vars}(\sigma)$; let X be the first variable in the numbering π_ω that is not in $\text{vars}(\sigma)$. This implies that $\text{Pred}_{\pi_\omega}[X](\omega)$ is a sub-instantiation of σ. But by the definition of a supportive numbering, $\text{Pred}_{\pi_\omega}[X](\omega)$ supports X. This means σ supports X as well, contradicting our assumption.

Finally, we must show that T satisfies Definition 3.3(iii): each variable serves as a split variable exactly once on each non-truncated path from the root. Since X_T^σ is chosen from the variables that are not in $\text{vars}(\sigma)$, a path cannot contain more

than one node that splits on a given variable. To show that each non-truncated path from the root splits on each variable in $\mathcal{V}$, consider any such path $\sigma_1, \sigma_2, \ldots$. We constructed T so that every incomplete instantiation has at least one child, so this path cannot end with an incomplete instantiation; it must be infinite. First consider the case where the path contains a directly disallowed instantiation σ_i. Then every instantiation after σ_i in the path is also directly disallowed, so the split variables from σ_i onward are simply selected according to the global numbering π_g. The number of steps from σ_i to a node that instantiates any variable X is at most $\pi_g(X)$.

If no instantiation in the path is directly disallowed, then consider the infinite instantiation $\wedge_i \sigma_i$. We can construct a supportive numbering π for $\wedge_i \sigma_i$ by letting $\pi(X)$ be the depth of the node in $\sigma_1, \sigma_2, \ldots$ that splits on X. Because no instantiation on this path is directly disallowed, this numbering must not yield a zero factor. So because Γ respects its outcome space, we know $\wedge_i \sigma_i$ is achievable. Let ω be an outcome consistent with $\wedge_i \sigma_i$, and let π_ω be a supportive numbering for ω. Now assume for contradiction that there is some variable in $\mathcal{V}$ that the path $\sigma_1, \sigma_2, \ldots$ does not split on. Let X be the first such variable, according to the numbering π_ω. Then the path does split on every variable in the finite set $\text{Pred}_{\pi_\omega}[X]$, so the path contains an instantiation σ_i that instantiates all of $\text{Pred}_{\pi_\omega}[X]$. But because split variables in T are selected according to the global numbering π_g, σ_i can have at most $\pi_g(X)$ consecutive descendants that split on variables other than X. So some node in the path must split on X.

Thus, the tree constructed in this way is a supportive split tree for Γ. Since Γ respects its outcome space, it follows by Theorem 3.12 that Γ is well-defined. $\square$

3.5 Contingent Bayesian networks

In a partition-based model, we do not need to specify the parents of a random variable; we just specify probability distributions for the variable given a set of events that partition the outcome space. However, for some purposes we would like a more explicit representation: one that specifies directly the set of variables that X depends on and the conditions under which these dependencies are active. Contingent Bayesian networks (CBNs) provide such a representation.

3.5.1 Decision trees for defining partitions

Like many previous approaches [Boutilier *et al.*, 1996], CBNs use decision trees to represent context-specific dependencies. Decision trees are like split trees (Definition 3.3), but without the requirement that each non-truncated path must split on every variable.

Definition 3.13. *A* decision tree T *over a set of random variables* $\mathcal{V}$ *is a directed tree where each node is an instantiation on* $\mathcal{V}$*, such that:*

- *the root node is* $\top$*;*
- *each non-leaf node* σ splits on *a variable* $X_T^\sigma \notin \text{vars}(\sigma)$ *such that the children of* σ *are* $\{(\sigma; X_T^\sigma = x) : x \in \text{range}(X_T^\sigma|\sigma)\}$.

Figure 3.8 shows decision trees for the variables in the urn-and-balls scenario of Example 3.1, and Figure 3.9 does the same for the hurricane scenario of Example 3.2. In some of these trees, there are nodes with infinitely many children; our definition also allows decision trees to contain infinite paths.

An outcome ω *matches* a path θ in a decision tree if ω is consistent with every node (instantiation) on the path. Note that although Definition 3.13 ensures that

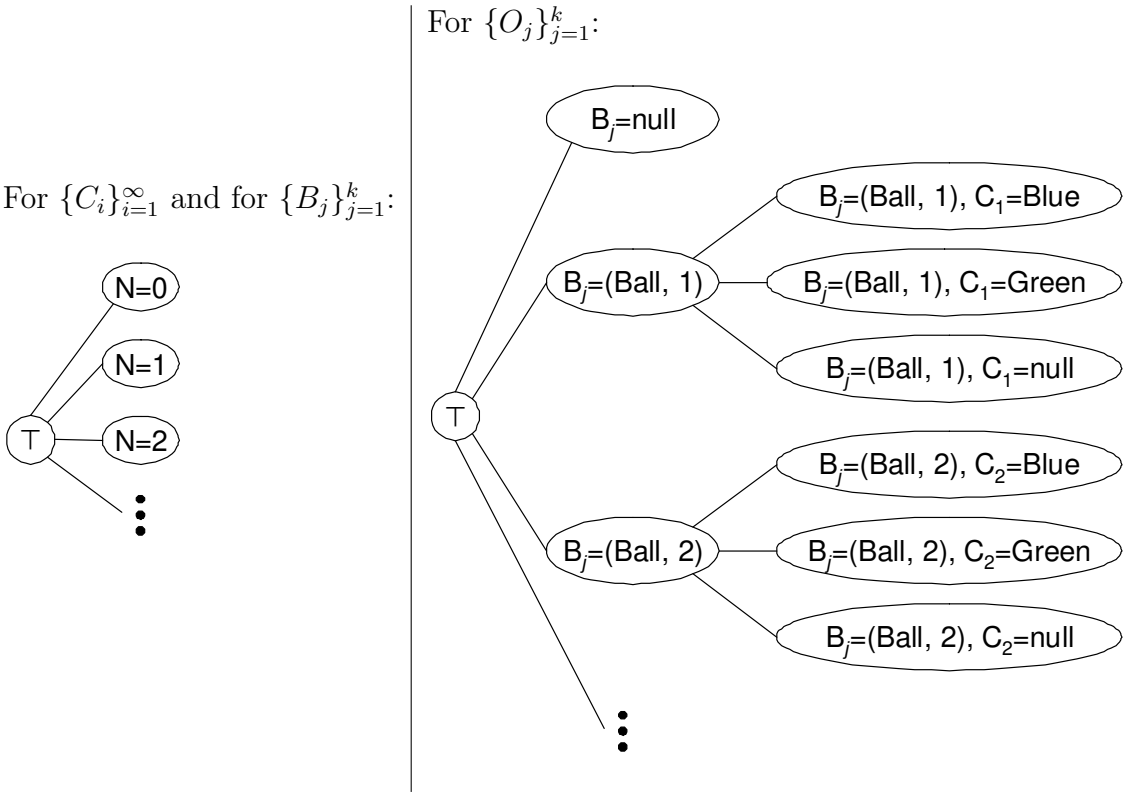


Figure 3.8: Decision trees for the variables in the urn-and-balls scenario with k draws. For the variable N, the decision tree consists of just a root node.

each instantiation in a decision tree is achievable, an infinite path may not have any matching outcomes (Definition 3.2 part (ii) does not apply because an infinite path in a decision tree does not necessarily split on every variable). If the set of outcomes that match θ is non-empty, we say θ is *achievable*. Recall that a *non-truncated* path is one that is infinite or ends at a leaf. The achievable, non-truncated paths starting from the root are mutually exclusive and exhaustive, so a decision tree defines a partition of Ω.

Definition 3.14. *Let $\mathcal{V}$ be a set of random variables defined on $(\Omega, \mathscr{F})$. The partition Λ_T defined by a decision tree T over $\mathcal{V}$ consists of a block of the form $\{\omega \in \Omega : \omega$ matches $\theta\}$ for each achievable, non-truncated path θ starting at the root*

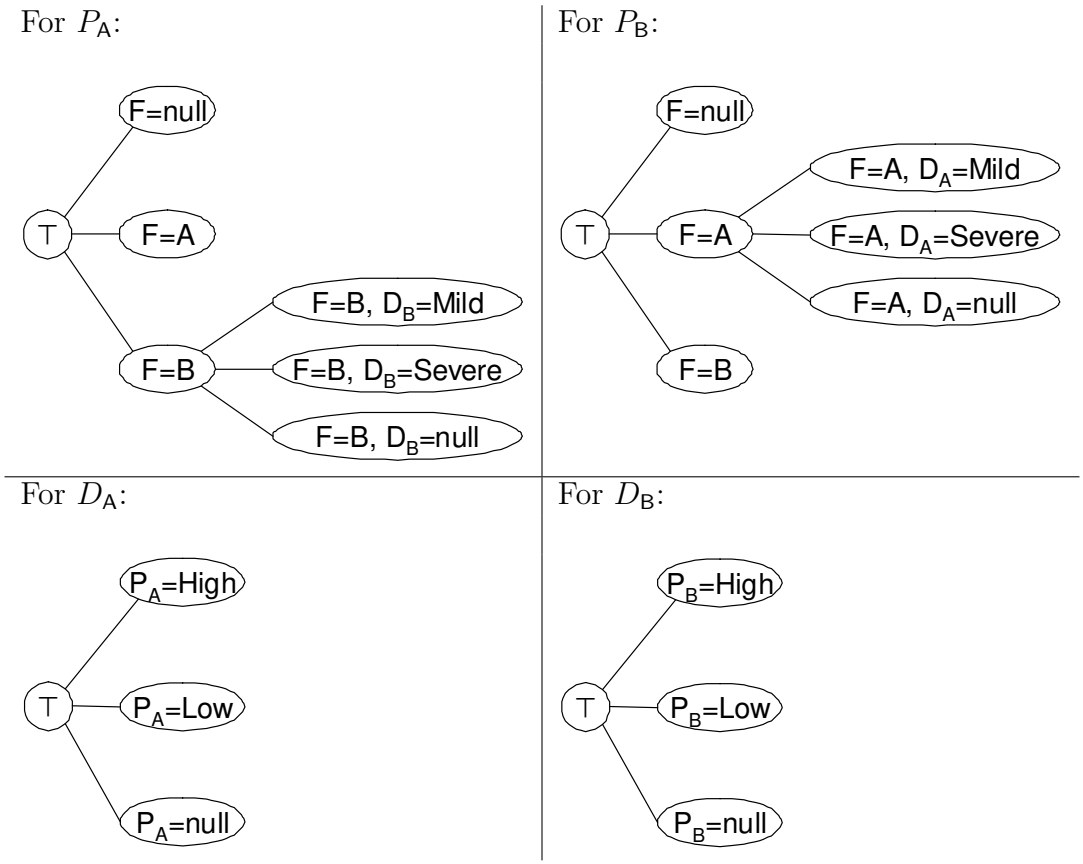


Figure 3.9: Decision trees for the variables in the hurricane scenario. For the variable F, the decision tree consists of just a root node.

of T.

For a tree with no infinite paths, the partition just consists of one block for each leaf node. For example, the decision tree for P_{A} in Figure 3.9 defines the partition:

$$\left\{ \begin{array}{c} \mathrm{ev}\left(F = \mathsf{null}\right),\ \mathrm{ev}\left(F = \mathsf{A}\right), \\ \mathrm{ev}\left(F = \mathsf{B}, D_{\mathsf{B}} = \mathsf{null}\right),\ \mathrm{ev}\left(F = \mathsf{B},\ D_{\mathsf{B}} = \mathsf{Mild}\right), \mathrm{ev}\left(F = \mathsf{B}, D_{\mathsf{B}} = \mathsf{Severe}\right) \end{array} \right\}$$

As another example, the decision tree for an ObservedColor node O_j in Figure 3.8

defines the partition:

$$\{\text{ev}\,(B_j = \mathsf{null})\} \cup \bigcup_{i=1}^{\infty} \left\{ \begin{array}{l} \text{ev}\,(B_j = (\mathsf{Ball}, i), C_i = \mathsf{null})\,, \\ \text{ev}\,(B_j = (\mathsf{Ball}, i), C_i = \mathsf{Blue})\,, \\ \text{ev}\,(B_j = (\mathsf{Ball}, i), C_i = \mathsf{Green}) \end{array} \right\}$$

Note that this partition for O_j is considerably finer than the partition we used for O_j in the PBM in Figure 3.6: that partition had only three blocks, one for each possible value of $\mathsf{TrueColor}(\mathsf{BallDrawn}(d))$. There is no decision tree that represents the fact that the events $(B_j = (\mathsf{Ball}, 1), C_1 = \mathsf{Blue})$ and $(B_j = (\mathsf{Ball}, 8), C_8 = \mathsf{Blue})$ are equivalent as far as O_j is concerned (for that we would need a decision graph [Chickering *et al.*, 1997]). However, the partition defined by the decision tree still captures a great deal of contingent structure, in that each block only specifies the color of the single ball that serves as the value of B_j. The advantage of representing the partition for a variable X as a decision tree (as opposed to in a more abstract form) is that it reveals which variables X depends on in which contexts.

3.5.2 CBN structures

Once we have specified which variables depend on which others in which contexts, we can represent these dependencies graphically in a *CBN structure*.

Definition 3.15. *Let $\mathcal{V}$ be a set of random variables defined on an outcome space $(\Omega, \mathscr{F})$. A* CBN structure *$\mathcal{G}$ for $\mathcal{V}$ is a directed graph where the set of nodes is $\mathcal{V}$, and each edge is labeled with an event in $\mathscr{F}$.*

An edge from W to X labeled with an event E, which we denote by $(W \to X \mid E)$, means that X depends on W when E occurs. As examples, Figure 3.10 shows a CBN structure for the urn-and-balls scenario, and Figure 3.11 shows a CBN structure for the hurricane scenario. In the urn-and-balls model, the edge $C_3 \to O_1$

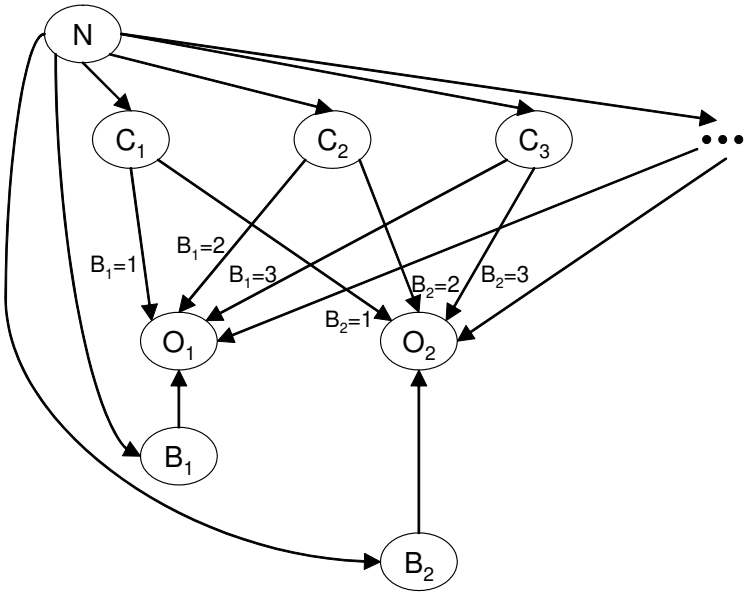


Figure 3.10: A CBN structure for the urn-and-balls scenario of Example 3.1.

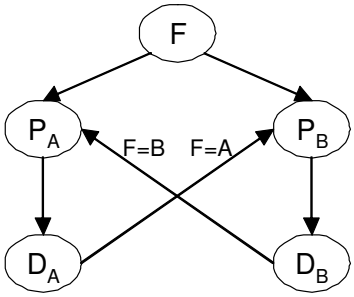


Figure 3.11: A CBN structure for the hurricane scenario of Example 3.2.

is labeled with the event $B_1 = 3$, meaning that $[\mathsf{ObsColor}]^\omega((\mathsf{Draw}, 1))$ depends on $[\mathsf{TrueColor}]^\omega((\mathsf{Ball}, 3))$ only when $[\mathsf{BallDrawn}]^\omega(\mathsf{Draw}, 1) = (\mathsf{Ball}, 3)$. In the hurricane model, the edge $D_B \to P_A$ is labeled with $F = B$, indicating that $[\mathsf{Prep}]^\omega(\mathsf{A})$ depends on $[\mathsf{Damage}]^\omega(\mathsf{B})$ only when $[\mathsf{First}]^\omega = \mathsf{A}$. Edges that are left unlabeled in our diagrams are formally labeled with the uninformative event Ω.

Given an outcome ω, an edge $(W \to X \mid E)$ is said to be *active* if $\omega \in E$, and *inactive* otherwise. A partial instantiation σ *activates* $(W \to X \mid E)$ if σ implies E (that is, $\mathrm{ev}(\sigma) \subseteq E$) and *deactivates* it if σ contradicts E (that is, $\mathrm{ev}(\sigma) \cap E = \varnothing$). For example, in Figure 3.10, the instantiation $B_1 = 3$ activates the edges $B_1 \to O_1$ and $(C_3 \to O_1 \mid B_1 = 3)$, and deactivates all the other edges into O_1. If an instantiation σ neither activates nor deactivates a given edge, we say this edge is *potentially active* given σ.

If we take a CBN structure $\mathcal{G}$ and retain just those edges that are active in an outcome ω (or activated by an instantiation σ), we get the *context-specific graph* $\mathcal{G}^\omega$ (or $\mathcal{G}^\sigma$). A variable W is an *active parent* of X given σ if it is a parent of X in $\mathcal{G}^\sigma$ — or equivalently, if an edge from W to X is activated by σ. The *active ancestors* of X given σ are those variables that are ancestors of X in $\mathcal{G}^\sigma$.

For each variable X in a CBN $\mathcal{B}$, we specify a decision tree $T_{\mathcal{B}}^X$, thus defining a partition $\Lambda_{\mathcal{B}}^X \triangleq \Lambda_{(T_{\mathcal{B}}^X)}$. To complete the parameterization, we also specify a CPD $c_{\mathcal{B}}^X$ for X given $\Lambda_{\mathcal{B}}^X$. However, the decision tree for X must respect the CBN structure in the following sense.

Definition 3.16. *A decision tree T* respects the CBN structure $\mathcal{G}$ *at X if for every node $\sigma \in T$ that splits on a variable W, there is an edge $(W \to X \mid E)$ in $\mathcal{G}$ that is activated by σ.*

For example, the decision trees for the urn-and-balls and hurricane scenarios in Figures 3.8 and 3.9 all respect the CBN structures in Figures 3.10 and 3.11. Note

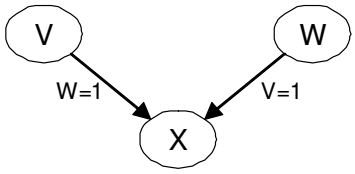


Figure 3.12: A CBN structure in which no decision tree can split on V or W while respecting the structure at X, despite the fact that V and W are parents of X.

that a decision tree T can respect a CBN structure at X even if does not split on all of X's parents. In fact, there are some CBN structures where it is impossible for a tree to split on certain parents while respecting the CBN structure. For example, consider the CBN structure in Figure 3.12. A decision tree that respects this structure at X cannot split on V except at a node where W has already been instantiated to 1, and it cannot split on W except where V has been set to 1. Thus, the tree cannot split on either variable. Only a trivial single-node decision tree respects this CBN structure at X. CBN structures of this kind are likely to be confusing, and should be avoided; however, our results still hold for them.

3.5.3 Structurally well-defined CBNs

We are now ready to give a formal definition of a CBN.

Definition 3.17. *Let $\mathcal{V}$ be a set of discrete random variables that is sufficient for $(\Omega, \mathscr{F})$. A* contingent Bayesian network *(CBN) $\mathcal{B}$ for $\mathcal{V}$ consists of a CBN structure $\mathcal{G}_\mathcal{B}$, and for each variable $X \in \mathcal{V}$:*

- *a decision tree $T^X_\mathcal{B}$ that respects $\mathcal{G}_\mathcal{B}$ at X and defines a countable partition $\Lambda^X_\mathcal{B} \triangleq \Lambda_{(T^X_\mathcal{B})}$;*
- *a CPD $c^X_\mathcal{B}$ for X given $\Lambda^X_\mathcal{B}$.*

It is clear that a CBN is a kind of PBM, since it defines a partition[3] and a CPD for each variable. Thus, our definitions for PBMs also specify what it means for a distribution to satisfy a CBN, and for a CBN to be well-defined. But the CBN representation provides additional insight: we can use graphical criteria to check when an instantiation supports a variable, and when an instantiation is self-supporting.

Proposition 3.14. *Let X be a variable in a CBN $\mathcal{B}$, and let σ be an achievable instantiation on $\mathcal{B}$'s variables. If the active parents of X given σ are all in* $\text{vars}(\sigma)$*, then σ supports X.*

Proof. Suppose all the active parents of X given σ are in $\text{vars}(\sigma)$. By Definition 3.17, we know the decision tree $T_{\mathcal{B}}^{X}$ respects $\mathcal{G}_{\mathcal{B}}$ at X. We claim that there is a unique non-truncated path from the root of $T_{\mathcal{B}}^{X}$ on which every node is a sub-instantiation of σ. We can construct this path $\tau_0, \tau_1, \ldots$ as follows. Let τ_0 be the root node $\top$, which is clearly a sub-instantiation of σ. Now suppose we have constructed the path up through τ_n. If τ_n is a leaf node, then $\tau_0, \ldots, \tau_n$ is a non-truncated path, and we are done. Otherwise, by Definition 3.16, the variable W that τ_n splits on is an active parent of X given τ_n. But by construction, τ_n is a sub-instantiation of σ, so W is an active parent of X given σ as well. Therefore, $W \in \text{vars}(\sigma)$. So because σ is achievable, the instantiation $(\tau; W = \sigma_W)$ is one of τ_n's children, and we can let this instantiation be τ_{n+1}. This is the only child of τ_n that is a sub-instantiation of σ, because each of τ_n's other children assigns a different value to W.

Let θ be the path constructed in this manner. Then every outcome $\omega \in \text{ev}(\sigma)$ matches θ. By Definition 3.14, there is a block in the partition $\Lambda_{T_{\mathcal{B}}^{X}}$ consisting of those outcomes that match θ. It follows that $\text{ev}(\sigma)$ is a subset of this block, so σ

[3]Definition 3.17 requires the decision trees to define countable partitions (ruling out decision trees with uncountably many paths from the root). And if $\mathcal{V}$ is a set of discrete random variables that is sufficient for $(\Omega, \mathscr{F})$, then any partition defined by a decision tree over $\mathcal{V}$ is measurable.

supports X. □

Proposition 3.15. *Let $\mathcal{B}$ be a CBN and σ be an instantiation on $\mathcal{B}$'s variables. If* $\mathrm{vars}(\sigma)$ *is an ancestral set in $\mathcal{G}_{\mathcal{B}}^{\sigma}$, then σ is self-supporting.*

Proof. Consider any $X \in \mathrm{vars}(\sigma)$. By the definition of an ancestral set, $\mathrm{vars}(\sigma)$ includes the parents of X in $\mathcal{G}_{\mathcal{B}}^{\sigma}$. So by Proposition 3.14, σ supports X. □

We can also use graphical criteria to determine whether a CBN is well-defined. Theorem 3.12 Our main theorem on well-definedness of PBMs is Theorem 3.12; it requires that the PBM have a supportive split tree. Theorem 3.13 tells us that a PBM has a supportive split tree if each of its outcomes has a supportive numbering. Conveniently, in the CBN case, we can assure ourselves that an outcome ω has a supportive numbering by considering certain properties of $\mathcal{G}_{\mathcal{B}}^{\omega}$. In the following lemma, recall that an *infinite receding chain* is an infinite sequence of nodes $X_1 \leftarrow X_2 \leftarrow X_3 \leftarrow \cdots$ such that for each n, X_{n+1} is a parent of X_n.

Lemma 3.16. *Let ω be an outcome in a CBN $\mathcal{B}$. The following two conditions are equivalent and imply that ω has a supportive numbering:*

i. in $\mathcal{G}_{\mathcal{B}}^{\omega}$, there are no cycles and no infinite receding chains, and every variable has finitely many parents;

ii. in $\mathcal{G}_{\mathcal{B}}^{\omega}$, there are no cycles and every variable has finitely many ancestors.

Proof. The equivalence of conditions (i) and (ii) follows from Prop. 2.6, which says that in a directed graph, every variable has finitely many ancestors if and only if there are no infinite receding chains and every variable has finitely many parents. Also, by Theorem 2.4, condition (ii) implies that $\mathcal{G}_{\mathcal{B}}^{\omega}$ has a topological numbering. Let $X_0, X_1, \ldots$ be an enumeration of $\mathcal{V}$ according to some topological numbering of $\mathcal{G}_{\mathcal{B}}^{\omega}$.

We will show that $X_0, X_1, \ldots$ is a supportive numbering for ω. For any $n < |\mathcal{V}|$, let σ be the instantiation of $X_0, \ldots, X_{n-1}$ that agrees with ω. Then the edges activated by σ in $\mathcal{B}$ are a subset of those that are active given ω, so the parents of X_n in $\mathcal{G}_{\mathcal{B}}^{\sigma}$ are all in $\{X_0, \ldots, X_{n-1}\}$. Therefore, by Proposition 3.14, σ supports X_n. □

Lemma 3.16 deals with the context-specific graph for a particular outcome. However, we can use it to derive a graphical criterion that looks directly at the CBN structure $\mathcal{G}_{\mathcal{B}}$. We call a set of edges in $\mathcal{G}_{\mathcal{B}}$ *consistent* if the events on the edges have a non-empty intersection: that is, if there is some outcome that makes all the edges active.

Theorem 3.17. *Suppose a CBN $\mathcal{B}$ over random variables $\mathcal{V}$ respects its outcome space and satisfies the following:*

(A1) No consistent path in $\mathcal{G}_{\mathcal{B}}$ forms a cycle.

(A2) No consistent path in $\mathcal{G}_{\mathcal{B}}$ forms an infinite receding chain.

(A3) No variable $X \in \mathcal{V}$ has an infinite, consistent set of incoming edges in $\mathcal{G}_{\mathcal{B}}$.

Then every outcome in $\mathcal{B}$ has a supportive numbering, and $\mathcal{B}$ is well defined.

Proof. Consider any outcome $\omega \in \Omega$. By conditions A1-A3 and the definition of a consistent set of edges, $\mathcal{G}_{\mathcal{B}}^{\omega}$ contains no cycles, no infinite receding chains, and no infinite parent sets. So condition (i) of Lemma 3.16 is satisfied, and ω has a supportive numbering. Since this is true for all outcomes, Theorem 3.13 implies that $\mathcal{B}$ has a supportive split tree; then Theorem 3.12 implies that $\mathcal{B}$ is well defined. □

A CBN that satisfies the conditions of Theorem 3.17 is said to be *structurally well-defined*. If a CBN has a finite set of variables, we can check the conditions directly. For instance, the hurricane CBN in Figure 3.11 is structurally well-defined:

although it contains a cycle, the cycle is not consistent. The urn-and-balls CBN in Figure 3.10) has infinitely many nodes, so we cannot check mechanically that it is structurally well-defined. However, it is clear from the representation in Figure 3.10 that the CBN contains no cycles or infinite receding chains. Also, although the ObsColor nodes O_1 and O_2 have infinitely many incoming edges, the labels on these edges ensure that exactly one of them is active in each outcome. So this CBN is also structurally well-defined.

We conclude this subsection with a lemma about the context-specific graphs that can be derived from structurally well-defined CBNs (the lemma will be used in Chapter 5). These graphs always have topological numberings. To understand this, it is important to remember that an instantiation σ *activates* an edge $(X \to Y \mid E)$ just when σ entails E. If σ neither entails nor contradictes E, then $(X \to Y \mid E)$ is not active given σ.

Lemma 3.18. *If a CBN $\mathcal{B}$ is structurally well-defined, each context-specific graph $\mathcal{G}^\omega_\mathcal{B}$ for an outcome $\omega \in \Omega$, or $\mathcal{G}^\sigma_\mathcal{B}$ for an achievable instantiation σ, has a topological numbering.*

Proof. First consider any outcome $\omega \in \Omega$. By the conditions in Theorem 3.17 and the definition of a consistent set of edges, $\mathcal{G}^\omega_\mathcal{B}$ contains no cycles, infinite receding chains, or infinite parent sets. So by Proposition 2.6 and Theorem ̊thm:finite-anc-topo-num, it follows that $\mathcal{G}^\omega_\mathcal{B}$ has a topological numbering.

Now consider an achievable instantiation σ. Because σ is achievable, there is some outcome $\omega \in \mathrm{ev}\,(\sigma)$. The set of edges activated by σ is a subset of those active in ω. So $\mathcal{G}^\sigma_\mathcal{B}$ is a subgraph of $\mathcal{G}^\omega_\mathcal{B}$. Thus because $\mathcal{G}^\sigma_\mathcal{B}$ has a topological numbering, $\mathcal{G}^\sigma_\mathcal{B}$ has that same topological numbering as well. □

3.5.4 CBNs as implementations of PBMs

In a PBM, we specify an arbitrary partition for each variable; in CBNs, we restrict ourselves to partitions generated by decision trees. But given any partition Λ, we can construct a decision tree T that yields a partition at least as fine as Λ—that is, such that each block $\lambda \in \Lambda_T$ is a subset of some $\lambda' \in \Lambda$. In the worst case, every path starting at the root in T will need to split on every variable. Thus, every PBM is *implemented* by some CBN, in the following sense:

Definition 3.18. *A CBN* $\mathcal{B}$ implements *a PBM* Γ *over the same set of variables* $\mathcal{V}$ *if, for each variable* $X \in \mathcal{V}$*, each block* $\lambda \in \Lambda^X_{cbn}$ *is a subset of some block* $\lambda' \in \Lambda^X_\Gamma$*, and* $c^X_\mathcal{B}(x, \lambda) = c^X_\Gamma(x, \lambda')$ *for all* $x \in \text{range}(X)$*.*

Proposition 3.19. *If a structurally well-defined CBN* $\mathcal{B}$ *implements a PBM* Γ*, then* Γ *is also well-defined, and* $\mathcal{B}$ *and* Γ *are satisfied by the same unique distribution.*

Proof. The first thing we show is that if an instantiation σ supports a variable X in $\mathcal{B}$, then σ supports X in Γ as well. To see this, note that if σ supports X in $\mathcal{B}$, then there is a block $\lambda \in \Lambda^\mathcal{B}_X$ such that $\text{ev}(\sigma) \subseteq \lambda$. But then because $\mathcal{B}$ implements Γ, there is a block $\lambda' \in \Lambda^\Gamma_X$ such that $\lambda \subseteq \lambda'$, and hence $\text{ev}(\sigma) \subseteq \lambda'$.

Therefore, if an instantiation is self-supporting in $\mathcal{B}$ then it is also self-supporting in Γ. Also, the stipulation in Definition 3.18 that $c^X_\mathcal{B}(x, \lambda) = c^X_\Gamma(x, \lambda')$ implies that $\mathcal{B}$ and Γ assign the same probabilities to all finite instantiations that are self-supporting in $\mathcal{B}$. So by Theorem 3.10, any probability measure that satisfies Γ also satisfies $\mathcal{B}$. Now since $\mathcal{B}$ is well-defined, there is exactly one probability measure P that satisfies it. This P is the only probability measure that could possibly satisfy Γ, because any other satisfiers of Γ would satisfy $\mathcal{B}$ as well.

So we just have to show that Γ is well-defined, and hence has some satisfier. Our observations so far imply (by Definition 3.11) that if π is a supportive numbering for

an outcome ω under $\mathcal{B}$, then it is also a supportive numbering for ω under Γ. Given that $\mathcal{B}$ is structurally well-defined, every outcome has a supportive numbering under $\mathcal{B}$, and hence under Γ as well. So by Theorem 3.12, Γ is also well-defined. □

Proposition 3.19 gives us a way to show that a PBM Γ is well-defined: construct a CBN $\mathcal{B}$ that implements Γ, and then use Theorem 3.17 to show that $\mathcal{B}$ is structurally well-defined. However, there may be multiple ways to implement a PBM as a CBN. As the following example illustrates, even if some of these implementations are structurally well-defined, others may not be.

Example 3.3. *Consider predicting who will go to a weekly book group meeting. Suppose it is usually Bob's responsibility to prepare questions for discussion, but if a historical fiction book is being discussed, then Alice prepares questions. In general, Alice and Bob each go to the meeting with probability 0.9. However, if the book is historical fiction and Alice isn't going, then the group will have no discussion questions, so the probability that Bob bothers to go is only 0.1. Similary, if the book is not historical fiction and Bob isn't going, then Alice's probability of going is 0.1. We will use H, G_A and G_B to represent the binary variables "historical fiction", "Alice goes", and "Bob goes".*

This scenario is most naturally represented by a PBM. The probability that Bob goes is 0.1 given $((H=1) \wedge (G_A=0))$ and 0.9 otherwise, so the partition for G_B has two blocks. The partition for G_A has two blocks as well.

The CBNs in Figure 3.13 both implement this PBM. There are no decision trees that yield exactly the desired partitions for G_A and G_B: the trees in Figure 3.13 yield three blocks instead of two. Because the trees on the two sides of the figure split on the variables in different orders, they respect CBN structures with different labels on the edges. The CBN on the left has a consistent cycle, while the CBN on the right is structurally well-defined.

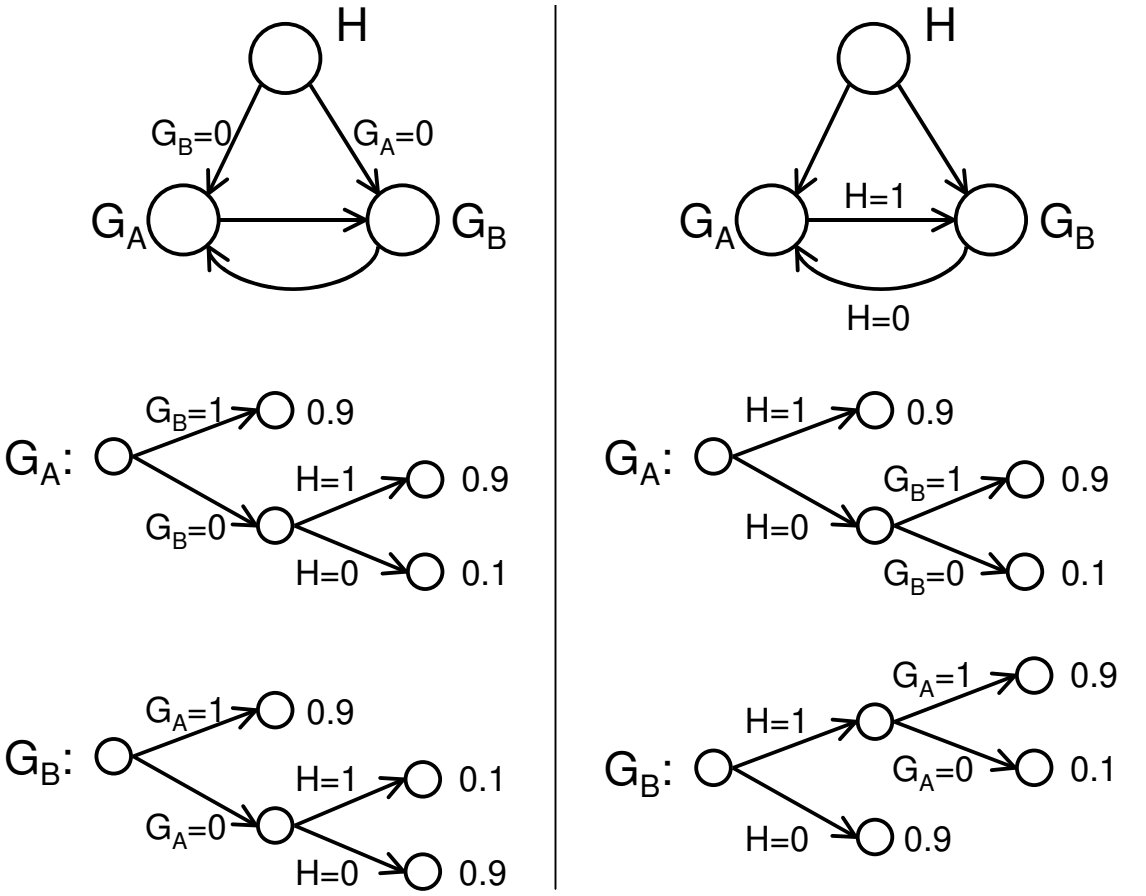


Figure 3.13: Two CBNs for Example 3.3, with decision trees for G_A and G_B. The numbers at the leaves of the trees are the probabilities of the value 1 given that leaf.

Thus, there may be multiple CBNs that implement a given PBM, and it may be that some of these CBNs are structurally well-defined while others are not. Even if we are given a well-defined PBM, it may be non-trivial to find a structurally well-defined CBN that implements it. Thus, algorithms that apply to structurally well-defined CBNs — such as the ones we define in Chapter 5 — cannot be extended easily to general PBMs.

Appendix 3.A Another factorization property

Theorem 3.10 tells us that if a PBM Γ has a supportive split tree, then a probability measure P satisfies Γ if and only if $P(\sigma)$ satisfies Equation 3.6 for every finite, self-supporting instantiation σ. In our IJCAI paper on BLOG [Milch *et al.*, 2005a], however, we use a different factorization property. Here we show that this alternative property is also necessary and sufficient for showing that a probability measure satisfies a PBM.

Proposition 3.20. *Let Γ be a PBM for a set of variables $\mathcal{V}$ defined on $(\Omega, \mathscr{F})$, and let P be a probability measure on $(\Omega, \mathscr{F})$. Suppose Γ respects its outcome space and has a supportive split tree. Then P satisfies Γ if and only if, for each finite instantiation σ on $\mathcal{V}$ that has a supportive numbering, there is a supportive numbering π such that:*

$$P(\sigma) = \prod_{X \in \text{vars}(\sigma)} c_\Gamma^X(\sigma_X, \lambda_\Gamma^X(\sigma[\text{Pred}_\pi[X]])) \tag{3.10}$$

Before proving this proposition, we prove the following lemma.

Lemma 3.21. *If an instantiation σ has a supportive numbering and is achievable, then $P(\sigma)$ satisfies Equation 3.10 for some supportive numbering if and only if it satisfies Equation 3.6.*

Proof. The only difference between these two equations is that in Equation 3.10, the partition block passed into X's CPD is $\lambda_\Gamma^X(\sigma[\text{Pred}_\pi[X]])$, while in Equation 3.6, it is $\lambda_\Gamma^X(\sigma)$. But σ is an achievable extension of $\sigma[\text{Pred}_\pi[X]]$, so we know $\lambda_\Gamma^X(\sigma) = \lambda_\Gamma^X(\sigma[\text{Pred}_\pi[X]])$. □

Proof of Prop. 3.20. First suppose P satisfies Γ, and consider any instantiation σ that has a supportive numbering. If σ is achievable, then it is self-supporting. So by Theorem 3.10, $P(\sigma)$ satisfies our earlier factorization equation Equation 3.6. Then

Lemma 3.21 implies that $P(\sigma)$ satisfies Equation 3.10 for some supportive numbering. Now suppose σ is not achievable. Then because Γ respects its outcome space, each supportive numbering of σ yields a zero factor. This implies that the right hand side of Equation 3.10 is zero for each supportive numbering. Also, since σ is not achievable and P is a probability measure on Ω, we know $P(\sigma) = 0$. So Equation 3.10 is satisfied.

For the converse, suppose $P(\sigma)$ satisfies Equation 3.10 for some supportive numbering of each instantiation σ that has such a numbering. Then by Lemma 3.21, $P(\sigma)$ satisfies Equation 3.6 for each instantiation σ that has a supportive numbering. In particular, if we let T be a supportive split tree for Γ, then Equation 3.6 is satisfied for every instantiation in T that has a supportive numbering. So by Theorem 3.10, P satisfies Γ. □

Appendix 3.B Supportive split trees revisited

Definition 3.10 says that a supportive split tree for a PBM is a split tree in which each non-leaf instantiation either supports its split variable, or is directly disallowed. This loophole for directly disallowed instantiations complicates the proofs of Lemmas 3.8 and 3.9. One might wonder if we are making our task unnecessarily difficult by not using a stricter definition of a supportive split tree that requires *all* nodes to support their split variables. It turns out that the loophole is necessary: without it, Theorem 3.13 — which guarantees the existence of a split tree when every outcome has a supportive numbering — would no longer hold. The following example describes a PBM in which every outcome has a supportive numbering, but for which there is no split tree in which every node supports its split variable.

Example 3.4. *Consider a variant of the urn-and-balls example where the blue and green balls are in separate urns. One urn contains a random, finite number of blue*

balls. There is an infinite sequence of draws: on each draw, a blue ball is picked randomly and then returned to the urn. The number of green balls in the other urn is equal to the index of the first draw that yields the same ball as an earlier draw. There must be some such draw in every outcome, because the sequence of draws is infinite and the number of distinct blue balls is finite.

As in the original urn-and-balls example, we will represent outcomes as model structures of a logical language with types Color, Ball, and Draw. The function symbols of our language are TrueColor and BallDrawn; we are not modeling the observed colors of the balls in this example. Our outcome space Ω consists of model structures ω where $[\mathsf{Color}]^\omega = \{\mathsf{Blue}, \mathsf{Green}\}$, $[\mathsf{Ball}]^\omega = \{(\mathsf{Ball}, \mathsf{Blue}, i)\}_{i=1}^n \cup \{(\mathsf{Ball}, \mathsf{Green}, i)\}_{i=1}^m$ for some natural numbers n and m, and $[\mathsf{Draw}]^\omega = \{(\mathsf{Draw}, j)\}_{j=0}^\infty$. We also constrain the outcome space so the function TrueColor is nonrandom. Specifically, for any ball (Ball, c, i), $\mathsf{TrueColor}((\mathsf{Ball}, c, i)) = c$. On this outcome space, we can define random variables N_{Blue}, N_{Green}, and $\{B_j\}_{j=0}^\infty$ as follows:

$$\begin{aligned} N_{\mathsf{Blue}}(\omega) &= |\{b \in [\mathsf{Ball}]^\omega \,:\, [\mathsf{TrueColor}]^\omega(b) = \mathsf{Blue}\}| \\ N_{\mathsf{Green}}(\omega) &= |\{b \in [\mathsf{Ball}]^\omega \,:\, [\mathsf{TrueColor}]^\omega(b) = \mathsf{Green}\}| \\ B_j(\omega) &= [\mathsf{BallDrawn}]^\omega((\mathsf{Draw}, j)) \end{aligned}$$

Our event space $\mathscr{F}$ is the σ-field generated by these random variables. Note that the only way a complete instantiation of these variables can be unachievable is if some BallDrawn variable B_j is assigned a value (Ball, c, i) such that $i > N_c$. So any unachievable complete instantiation has an unachievable sub-instantiation of size two. Thus, these variables are sufficient (in the sense of Definition 3.2) for the outcome space we have defined.

Figure 3.14 describes a PBM Γ over these variables. For brevity, we use the notation RepeatIndices(ω) to denote the set of natural numbers j such that for some

For the variable N_{Blue}:

$$\Lambda_{\Gamma}^{N_{\mathsf{Blue}}} = \{\Omega\}$$
$$c_{\Gamma}^{\Gamma} N_{\mathsf{Blue}}(n, \Omega) = \text{Poisson}[6](n)$$

For the variables $\{B_j\}_{j=0}^{\infty}$:

$$\Lambda_{\Gamma}^{B_j} = \{\{\omega \in \Omega \,:\, |\{b \in [\mathsf{Ball}]^{\omega} \,:\, [\mathsf{TrueColor}]^{\omega}(b) = \mathsf{Blue}\}| = n\}\}_{n \in \mathbb{N}}$$
$$c_{\Gamma}^{B_j}(b,\, \{\omega \in \Omega \,:\, |\{b \in [\mathsf{Ball}]^{\omega} \,:\, [\mathsf{TrueColor}]^{\omega}(b) = \mathsf{Blue}\}| = 0\}) = \begin{cases} 1 & \text{if } b = \mathsf{null} \\ 0 & \text{otherwise} \end{cases}$$
$$c_{\Gamma}^{B_j}(b,\, \{\omega \in \Omega \,:\, |\{b \in [\mathsf{Ball}]^{\omega} \,:\, [\mathsf{TrueColor}]^{\omega}(b) = \mathsf{Blue}\}| = n\}) = \begin{cases} 1/n & \text{if } b \in \{(\mathsf{Ball}, \mathsf{Blue}, 1), \ldots, (\mathsf{Ball}, \mathsf{Blue}, n)\} \\ 0 & \text{otherwise} \end{cases}, \text{ for } n > 0$$

For the variable N_{Green}:

$$\Lambda_{\Gamma}^{N_{\mathsf{Green}}} = \{\{\omega \in \Omega \,:\, \min\{j \,:\, j \in \text{RepeatIndices}(\omega)\} = j^*\}\}_{j^*=0}^{\infty}$$
$$c_{\Gamma}^{N_{\mathsf{Green}}}(m, \{\omega \in \Omega \,:\, \min\{j \,:\, j \in \text{RepeatIndices}(\omega)\} = j^*\}) = \begin{cases} 1 & \text{if } m = j^* \\ 0 & \text{otherwise} \end{cases}$$

Figure 3.14: Partitions and CPDs of a PBM Γ for Example 3.4.

$j' < j$, $[\mathsf{BallDrawn}]^{\omega}((\mathsf{Draw}, j)) = [\mathsf{BallDrawn}]^{\omega}((\mathsf{Draw}, j'))$. Note that because $[\mathsf{Ball}]^{\omega}$ is finite in every outcome $\omega \in \Omega$, the function $[\mathsf{BallDrawn}]^{\omega}$ cannot yield distinct values on an infinite sequence of draws. Thus RepeatIndices(ω) is non-empty for each $\omega \in \Omega$.

Let us consider what instantiations support each variable in this PBM. The variable N_{Blue} is supported by any instantiation, because $\Lambda_{\Gamma}^{N_{\mathsf{Blue}}}$ is the trivial partition. Each BallDrawn variable B_j is supported by any instantiation that includes N_{Blue}, because the distribution for B_j depends only on the number of blue balls. As to N_{Green}, it is supported by any instantiation σ that includes a prefix $B_0, \ldots, B_j$ of the BallDrawn variables such that $\sigma_{B_j} = \sigma_{B_{j'}}$ for some $j' < j$.

This PBM allows us to prove the follow proposition.

Proposition 3.22. *There is a PBM that supports its outcome space, and in which*

every outcome has a supportive numbering, but that has no split tree in which every node supports its split variable.

Proof. The PBM described in Figure 3.14, over the random variables and outcome space described earlier in this section, is such a PBM. To show that this PBM respects its outcome space (Definition 3.12), consider any instantiation σ that has a supportive numbering. If σ instantiates any $\mathsf{BallDrawn}$ variables, then it must also instantiate N_{Blue}. If σ instantiates each $\mathsf{BallDrawn}$ variable B_j to a value $(\mathsf{Ball}, \mathsf{Blue}, i)$ where $i \leq \sigma_{N_{\mathsf{Blue}}}$, then σ is achievable. And if σ instantiates any B_j to a value outside this set, then the CPDs for the B_j variables in Figure 3.14 make σ directly disallowed. So every instantiation that has a supportive numbering is either achievable or directly disallowed.

We now claim that every outcome has a supportive numbering under this PBM. This follows from our earlier conclusion that RepeatIndices(ω) is non-empty for each $\omega \in \Omega$. For any outcome ω, we can obtain a supportive numbering by taking N_{Blue} first, then $B_0, \ldots, B_{j^*}$ where $j^* = \min\{j : j \in \text{RepeatIndices}(\omega)\}$, then N_{Green}, then the remaining B_j variables in order by their indices.

Now we will show that this PBM has no split tree in which every node supports its split variable. Assume for contradiction that T is such a tree. Because N_{Green} is not supported unless some B_j variables are instantiated, and no B_j node is supported unless N_{Blue} is instantiated, the root node of T must split on N_{Blue}. We claim that T must have an infinite path starting at the root on which every node except the root splits on a $\mathsf{BallDrawn}$ variable, and the value assigned to the ith $\mathsf{BallDrawn}$ variable in the path is $(\mathsf{Ball}, \mathsf{Green}, i)$. Note that the path consisting of just the root node trivially satisfies this property. Now consider any n-node path with this property, and let σ be the last node in this path. If σ is the root node (*i.e.*, $n = 1$), then let σ' be an arbitrary child of σ. Otherwise, the property implies that σ splits on

a BallDrawn variable B_j, and this is the $(n-1)$th BallDrawn variable on the path to σ. Let $\sigma' = (\sigma; B_j = (\mathsf{Ball}, \mathsf{Green}, n-1))$. We know σ' is a child of σ in T because there are some outcomes consistent with σ in which $N_{\mathsf{Green}} \geq n-1$, and thus $(\mathsf{Ball}, \mathsf{Green}, n-1) \in \text{range}\,(B_j|\sigma)$. Furthermore, regardless of whether σ is the root node, σ' does not assign the same value to any two BallDrawn variables. Thus σ' does not support N_{Green}, so it must split on a BallDrawn variable. Thus the n-node path to σ can be extended to an $(n+1)$-node path to σ' with the stated property. So T contains an infinite sequence of paths $\theta_1, \theta_2, \ldots$ satisfying the property, such that θ_i is a proper prefix of θ_{i+1}. Taking the union of these paths yields an infinite path in which no node splits on N_{Green}, contradicting the assumption that T is a split tree (specifically, contradicting Definition 3.3(iii)). □

As guaranteed by Theorem 3.13, there is a tree that satisfies our weaker definition of a supportive split tree for this PBM. The CPDs given for the BallDrawn nodes B_j in Figure 3.14 assign zero probability to all non-blue balls. Thus, all instantiations that assign values of the form $(\mathsf{Ball}, \mathsf{Green}, j)$ to BallDrawn variables are directly disallowed. So we can construct a split tree in which such instantiations split on the variable N_{Green}, even if they do not support it. Exploiting this loophole for directly disallowed instantiations is the only way to construct a supportive split tree for this PBM.

Chapter 4

Bayesian Logic (BLOG)

We began the previous chapter by observing that standard Bayesian networks (BNs) are insufficient for representing certain scenarios involving unknown objects and relational uncertainty. We then introduced partition-based models (PBMs) and contingent Bayesian networks (CBNs): generalizations of BNs that explicitly represent the contingent nature of dependencies among variables. But on their own, these formalisms are not satisfactory languages for describing probabilistic models in machine-readable form.

The first shortcoming of PBMs and CBNs is that they do only part of the job of describing a probabilistic model. The first step in describing a probabilistic model is to specify the outcome space; often the second step is to define a set of random variables on that space. But a PBM or CBN applies to an outcome space and a set of variables that have already been specified. In fact, for the urn-and-balls example (Example 3.1) and the hurricane example (Example 3.2), we spent several paragraphs describing the outcome spaces and random variables that we used. In standard BNs, specifying the outcome space is not such a complex task: one simply specifies a range for each random variable, and the outcome space is the Cartesian

product of these ranges. But the task becomes more complicated when the outcomes are relational structures.

PBMs and CBNs also fall short in that they are propositional rather than first-order. That is, every variable is treated separately; there is no generalization across variables that represent the same attributes or relations on different objects. Of course, we did generalize across objects in our descriptions of PBMs: in Figure 3.6, we made general statements about classes of variables C_i, B_j, and O_j, rather than individual variables such as C_8 and B_3. We could not possibly have described the partition and CPDs for each variable separately because there are infinitely many of them in the urn-and-balls model. Similarly, in the CBN in Figure 3.10, we used an ellipsis to indicate an infinite sequence of TrueColor variables C_i. But we did not have any formal language for describing large or infinite models concisely.

Furthermore, our goal is not just to provide a language for describing PBMs or CBNs compactly, but also to facilitate the process of modeling scenarios with unknown objects. In such scenarios, it is often far from obvious how to formalize the possible outcomes: should the outcomes include objects that are not directly observable? Should they just include partitions of the observable objects? Choosing the set of random variables in a PBM or CBN raises similar questions: should variables be indexed by unobservable objects? Should we include variables that count the number of objects of some type, or variables that indicate whether particular objects exist? If we wish to apply the theory developed in the previous chapter, then these choices are subject to certain constraints. The random variables must be sufficient for the outcome space (Definition 3.2): intuitively, all measurable sets must be expressible in terms of instantiations of the random variables. And the probability model must respect the outcome space (Definition 3.12) so that unachievable instantiations receive zero probability. We would like a modeling language that reduces the need for the modeler to worry about these conditions for each new domain.

This chapter introduces Bayesian logic (BLOG), a language that concisely specifies probability distributions over relational structures with varying sets of objects. The relational structures, or *possible worlds*, are model structures of a typed first-order logical language (see Section 2.1). There are two complementary ways of understanding BLOG semantics. The first is that the BLOG model defines a randomized generative process that constructs a world step by step. The generative steps are of two kinds: steps that add a random number of objects to the world, and steps that set the value of a function on a tuple of arguments. The more formal and declarative perspective on BLOG semantics is that a BLOG model defines a PBM whose outcome space is the set of possible worlds. The variables of this PBM are defined implicitly by the BLOG model, in such a way that they are always sufficient for the outcome space. Also, the PBM defined by a BLOG model never assigns positive probability to unachievable instantiations.

We begin in Section 4.1 by presenting BLOG models for four example scenarios. Two of these are the urn-and-balls scenario (Example 3.1) and the hurricane scenario (Example 3.2) from the previous chapter. The other two are more realistic: one involves disambiguating bibliographic citations, and the other involves tracking aircraft. We explain these BLOG models at an intuitive level, in terms of the generative world-construction processes that they describe. In Section 4.2, we give a more formal and systematic explanation of BLOG syntax. Section 4.3 gives declarative semantics for BLOG in terms of PBMs, showing that every BLOG model defines a PBM that respects its outcome space.

In Section 4.4, we discuss how to evaluate expressions in a BLOG model given a partial instantiation of the model's random variables. This section introduces *object generation graphs*, which help to limit the set of objects we iterate over when evaluating expressions, and thus prevent infinite loops in certain cases. Finally, Section 4.5 discusses automatically checkable conditions under which a BLOG model

is guaranteed to define a unique distribution. Along the way, it defines the *canonical contingent BN* for a BLOG model, which plays a role in our inference algorithms in Chapter 5.

4.1 Examples

Our first BLOG model is for the urn-and-balls scenario of Example 3.1. Recall that in this scenario, we have an urn containing a finite but unknown number of balls, each of which is either blue or green. We repeatedly draw a ball from the urn, observe its color (with some probability of error), and return it to the urn. Given these observations, we may want to obtain a posterior probability distribution on the number of balls in the urn, or find the posterior probability that the first and second draws yielded the same ball.

Figure 4.1 shows a BLOG model for this scenario. The first 6 lines of this model define the particular logical language whose model structures are to serve as possible worlds. Line 1 says that the language contains three types: Color, Ball, and Draw. Lines 2–4 say that the language includes three function symbols, TrueColor, BallDrawn and ObsColor, which are all random: their interpretations vary from world to world. The argument types and return types of these functions are given in a syntax reminiscent of C or Java. Line 5 asserts that two colors, denoted by the constant symbols Blue and Green, are guaranteed to exist in all possible worlds; line 6 makes a similar assertion about draws.

In the hypothetical stochastic process for constructing a possible world, we begin with just the colors and draws that are guaranteed to exist. Then the *number statement* on line 7 says that we add a random number of balls, sampled according to a Poisson distribution with mean 6. Line 8 says that for each ball b, TrueColor(b) is sampled according to a probability table that puts probability 0.5 on Blue and 0.5

```
1   type Color; type Ball; type Draw;

2   random Color TrueColor(Ball);
3   random Ball BallDrawn(Draw);
4   random Color ObsColor(Draw);

5   guaranteed Color Blue, Green;
6   guaranteed Draw Draw1, Draw2, Draw3, Draw4;

7   #Ball ~ Poisson[6]();

8   TrueColor(b) ~ TabularCPD[[0.5, 0.5]]();

9   BallDrawn(d) ~ Uniform({Ball b});

10  ObsColor(d)
11      if (BallDrawn(d) != null) then
12          ~ TabularCPD[[0.8, 0.2], [0.2, 0.8]]
13              (TrueColor(BallDrawn(d)));
```

Figure 4.1: BLOG model for balls in an urn (Example 3.1) with four draws.

on Green (the ordering of the probabilities in the vector $[0.5, 0.5]$ corresponds to the order in which Blue and Green were introduced on line 5). Next, for each draw d, line 9 tells us to sample the value of BallDrawn(d) uniformly from the set of balls that exist. Finally, lines 10–13 tell us how to sample the observed color for each ball d. If BallDrawn(d) $\neq$ null, then ObsColor(d) is sampled according to a tabular CPD that conditions on TrueColor(BallDrawn(d)). The case where BallDrawn(d) = null occurs only when the number of balls in the world happens to be zero; in this case, ObsColor(d) gets a default value of null. The statements on lines 8 and 10–13 are called *dependency statements*.

This generative process defines a prior probability distribution over possible worlds. If we condition on observed values for certain random variables, such

```
1   type City; type PrepLevel; type DamageLevel;

2   random City First;
3   random PrepLevel Prep(City);
4   random DamageLevel Damage(City);

5   guaranteed City A, B;
6   guaranteed PrepLevel High, Low;
7   guaranteed DamageLevel Severe, Mild;

8   First ~ Uniform({City c});

9   Prep(c)
10      if First = c then ~ TabularCPD[[0.5, 0.5]]
11      else ~ TabularCPD[[0.9, 0.1], [0.1, 0.9]](Damage(First));

12  Damage(c) ~ TabularCPD[[0.2, 0.8], [0.8, 0.2]](Prep(c));
```

Figure 4.2: A BLOG model for the hurricane scenario (Example 3.2).

as (ObsColor(Draw1) = Blue, ObsColor(Draw2) = Blue), then a query event such as BallDrawn(Draw1) = BallDrawn(Draw2) has a well-defined posterior probability. Computing posterior probabilities is the task of the inference algorithms that we discuss in Chapter 5.

Our next BLOG model, shown in Figure 4.2, is for the hurricane scenario of Example 3.2. The premise for this scenario is that a hurricane is going to strike two cities, A and B, but it is unknown which city will be struck first. The level of damage that each city sustains depends on its level of preparations; also, the level of preparations in the second city to be hit depends on the level of damage in the first. The logical language used for this scenario has three types, City, PrepLevel and DamageLevel, and three random functions, First, Prep and Damage. Note that First is a random zero-ary function; in other words, it is a constant symbol whose denotation is random. In this model, the only objects that exist are guaranteed objects: the

cities A and B, the preparation levels High and Low, and the damage levels Severe and Mild.

Line 8 says that the denotation of First is chosen uniformly at random from the set of cities. Lines 9–11 give the probability model for Prep(c): if c is the first city hit, then Prep(c) has a uniform 0.5–0.5 prior distribution; otherwise, Prep(c) depends on Damage(First) through a tabular CPD. Finally, line 12 says that Damage(c) depends on Prep(c) through a certain CPD.

Our next example is a simplified version of the bibliographic citation matching problem [Lawrence *et al.*, 1999b; Pasula *et al.*, 2003].

Example 4.1. *Suppose we have collected a set of citation strings from the works cited sections of online papers. The citations may be in many different formats; they may use various abbreviations; and they may contain typographical errors. The task is to construct a database containing exactly one record for each publication that is cited and each researcher who is mentioned in these citations. This database should specify the correct names and titles of the entities, as well as the mapping from publications to their authors.*

A possible world for this scenario includes both the citations, which we observe, and the researchers and publications that underlie them. We can think of these worlds as model structures of a first-order language with types Researcher, Publication, Citation and String (the String type is actually built into BLOG). This language includes a function Name that maps researchers to strings; a function Title that maps publications to strings, and a function Author that maps publications to researchers. We also have a function PubCited that maps citations to the publications they refer to, and a function Text that maps citations to their observed text strings.

Figure 4.3 shows a BLOG model for this example, based on the model in [Pasula

```
1   type Researcher; type Publication; type Citation;

2   random String Name(Researcher);
3   random String Title(Publication);
4   random NaturalNum NumAuthors(Publication);
5   random Researcher NthAuthor(Publication, NaturalNum);
6   random Publication PubCited(Citation);
7   random String Text(Citation);

8   guaranteed Citation Cit1, Cit2, Cit3, Cit4;

9   #Researcher ~ NumResearchersPrior();
10  #Publication ~ NumPubsPrior();

11  Name(r) ~ NamePrior();

12  Title(p) ~ TitlePrior();
13  NumAuthors(p) ~ NumAuthorsPrior();
14  NthAuthor(p, n)
15      if n < NumAuthors(p) then ~ Uniform({Researcher r});

16  PubCited(c) ~ Uniform({Publication p});
17  Text(c) ~ NoisyCitationGrammar
18              (Title(PubCited(c)),
19               {n, Name(NthAuthor(PubCited(c), n))
20                for NaturalNum n :  n < NumAuthors(PubCited(c))});
```

Figure 4.3: BLOG model for Example 4.1 with four observed citations.

et al., 2003]. The model makes the simplifying assumption that in the academic field we are dealing with, there is a pool of researchers who are all equally likely to write papers. Line 9 describes the step that generates these researchers; the number of researchers is chosen from a CPD called `NumResearchersPrior`. Similarly, we assume that there is a pool of publications (generated by line 10) that are all equally likely to be cited. The name of each researcher and the title of each publication are sampled from certain prior distributions (lines 11 and 12). Each publication

has some number of authors; each author is sampled uniformly from the pool of researchers (lines 14–15). Then, for each citation, the publication cited is chosen uniformly from the pool of publications (line 16). The text of a citation c depends on the title of $\mathsf{PubCited}(c)$ and the names of its authors.

Note that under this model, a possible world may contain publications that are not referred to by any citation (and also researchers that are not authors of any publication). This is realistic: a finite collection of citations may not cover all the publications that could be cited. This model also ensures that questions such as, "What is the probability that an author named "Leslie Kaelbling" has written a paper that is not cited in our collection?" have well-defined answers. One might worry that if our queries are just about cited publications, then having a potentially large number of uncited publications in our possible worlds would make inference slower than necessary. However, as we will see in Chapter 5, there are inference algorithms that simply ignore the attributes of such irrelevant objects.

For our final example, we move from citations to radar blips. We describe a fairly standard version of the multitarget tracking problem [Bar-Shalom and Fortmann, 1988].

Example 4.2. *An unknown number of aircraft exist in some volume of airspace. An aircraft's state (position and velocity) at each time step depends on its state at the previous time step. We observe the area with radar: aircraft may appear as identical blips on a radar screen. Each blip gives the approximate position of the aircraft that generated it. However, some blips may be false detections, and we may not get a blip for every aircraft at every time step. The questions we would like to answer include: What aircraft exist? What are their trajectories? Are there any aircraft that have not been observed?*

The possible worlds of this scenario specify the trajectories of all the aircraft over

time, as well as the time stamps and locations of all radar blips that appear. We will model the states of the aircraft at an infinite sequence of time steps $0, 1, 2, \ldots$, although events at "future" time steps after our last observation will be irrelevant for inference (unless we explicitly ask queries about the future). These possible worlds can be represented as model structures of a first-order language with types Aircraft and Blip, as well as built-in types that represent natural numbers (time steps) and real vectors. The trajectories of the aircraft are represented by a function $\mathsf{State}(a, t)$ which maps aircraft a and natural numbers t to six-dimensional real vectors (three dimensions for the aircraft's position and three for velocity). A function MeasuredPos from blips to $\mathbb{R}^3$ represents the range, azimuth, and elevation that are measured for each blip. There is also a function Source that maps each blip to the aircraft that generated it; false detection blips have Source values of null. Finally, a function Time maps each blip to the time when it appeared (a natural number).

The BLOG model for this scenario (Figure 4.4) describes the following process: first, sample the number of aircraft in the area. Then, for each time step t (starting at $t = 0$), choose the state of each aircraft given its state at time $t - 1$. Also, for each aircraft a and time step t, possibly generate a radar blip b with $\mathsf{Source}(b) = a$ and $\mathsf{Time}(b) = t$. Whether a blip is generated or not depends on the state of the aircraft. Also, at each step t, generate some false-alarm blips b' with $\mathsf{Time}(b') = t$ and $\mathsf{Source}(b') = \mathsf{null}$. Note that the *origin functions* Source and Time are set on each blip when it is generated; they are not set in separate random sampling steps. Finally, sample the position for each blip given the true state of its source aircraft (or using a default distribution for a false-alarm blip).

This generative process is different from those in the previous examples in that the objects of each type are not all created in a single step. Here, there are many generative steps that add blips to the world: one for each aircraft a and each time t, plus additional steps that generate false-alarm blips. Intuitively, objects are gen-

```
1   type Aircraft; type Blip;

2   random R6Vector State(Aircraft, NaturalNum);
3   random R3Vector MeasuredPos(Blip);

4   origin Aircraft Source(Blip);
5   origin NaturalNum Time(Blip);

6   #Aircraft ~ NumAircraftPrior();

7   State(a, t)
8       if t = 0 then ~ InitState()
9       else ~ StateTransition(State(a, Pred(t)));

10  #Blip(Source = a, Time = t) ~ DetectionCPD(State(a, t));
11  #Blip(Time = t) ~ NumFalseAlarmsPrior();

12  MeasuredPos(b)
13      if (Source(b) = null) then ~ FalseAlarmDistrib()
14      else ~ MeasurementCPD(State(Source(b), Time(b)));
```

Figure 4.4: BLOG model for Example 4.2.

erating other objects. Furthermore, the numbers of blips generated can depend on the values of the State function, so the probability model does not easily decompose into a portion that defines a distribution for what objects exist and a portion that defines distributions for attribute values given object existence. Uncertainty about what objects exist is fully integrated with uncertainty about attributes and relations.

4.2 Syntax

This section provides a more formal definition of BLOG syntax. A BLOG model consists of a sequence of statements, separated by semicolons. Thus, for example, line 1 of Figure 4.1 contains three statements, while lines 10–13 of that figure constitute

Type symbol	Extension	Examples of built-in constants
Boolean	{true, false}	`true`, `false`
NaturalNum	$\mathbb{N}$	`0`, `1`, `2`
Real	$\mathbb{R}$	`3.14`, `-2.7`, `1.0`
RkVector (for $k \geq 2$)	$\mathbb{R}^k$	
String	Finite character strings	`""`, `"hello"`, `"R2-D2"`

Table 4.1: Built-in types, their extensions, and some illustrative built-in constant symbols denoting objects of each type. The Real type and the RkVector types are not included in discrete BLOG.

a single statement. There are six kinds of statements in BLOG: type declarations, function declarations, guaranteed object statements, nonrandom function definitions, dependency statements, and number statements. A BLOG model M defines a typed first order language $\mathcal{L}_M$, a set of possible worlds Ω_M, and (if it is well-defined) a probability measure P_M on Ω_M.

4.2.1 Built-in types and functions

The language defined by a BLOG model always includes a set of built-in types, summarized in Table 4.1. Each built-in type has a fixed extension in all possible worlds: for instance, the extension of NaturalNum is always the natural numbers. For most built-in types, there are also built-in constant symbols that denote objects of that type. Numerals without decimal points are constant symbols denoting objects of type NaturalNum; numerals with decimal points denote objects of type Real; strings enclosed in double quotes denote objects of type String. The interpretations of these constant symbols are fixed across all possible worlds.

Types representing real-valued vectors are a special case. For every natural number $k \geq 2$, there is a built-in type RkVector; examples include R2Vector, R3Vector, etc. Vectors are denoted not by constant symbols, but by terms using built-in functions, which we will introduce below.

Function symbol	Arguments	Return type	Shorthand
Succ	NaturalNum n	NaturalNum	
Pred	NaturalNum n	NaturalNum	
Sum	NaturalNum n, NaturalNum m	NaturalNum	n + m
Diff	NaturalNum n, NaturalNum m	NaturalNum	n - m
LessThan	NaturalNum n, NaturalNum m	Boolean	n < m
GreaterThan	NaturalNum n, NaturalNum m	Boolean	n > m
LessThanOrEqual	NaturalNum n, NaturalNum m	Boolean	n <= m
GreaterThanOrEqual	NaturalNum n, NaturalNum m	Boolean	n >= m
ConstructRkVector	Real r_1, ..., Real r_k	RkVector	[r_1, ..., r_k]
Concat	String s_1, String s_2	String	

Table 4.2: Built-in functions, their argument types (with variables standing for the arguments) and return types, and any special syntax that is used for them.

The Real type and the RkVector types have uncountable extensions, but our development of PBMs in Chapter 3 is limited to discrete variables. This discrepancy reflects a difference between our implemented BLOG engine, which supports models with continuous variables, and our theoretical results, which do not yet apply to such models. We discuss continuous types here to show that BLOG syntax is general enough to handle them. However, our formal results in this thesis apply only to *discrete BLOG*, a version of BLOG without the Real and RkVector types.

We now move on to BLOG's built-in functions, listed in Table 4.2. Like the built-in types, these functions are included in the language defined by each BLOG model. The built-in functions include Succ and Pred, which yield the successor and predecessor of a natural number (Pred(0) returns null). There are also built-in functions Sum and Diff on natural numbers; Diff(n, m) evaluates to null when n is less than m. Standard comparison functions such as LessThan are also built in. Furthermore, rather than requiring modelers to write terms such as `LessThan(3, 4)`, BLOG supports infix operators as a shorthand. Thus, one can use terms such as `3 < 4`. Next, for each natural number $k \geq 2$, BLOG includes a function ConstructRkVector that takes real numbers $r_1, \ldots, r_k$ and returns the vector $(r_1, \ldots, r_k)$. The shorthand way

to invoke such a function is to include k real-valued terms in square brackets: for example, `[0.2, 1.9, -2.0]`. The advantage of treating these expressions as function applications rather than just large constant symbols is that the vector elements can be represented by arbitrary terms, not just numerals. Finally, there is a built-in function Concat that returns the concatenation of two strings.

BLOG currently has no built-in constructs for referring to lists of objects, except for vectors of real numbers. As we saw in the citation model (Figure 4.3, we can represent lists using functions that take a natural number as an argument, such as NthAuthor(p, n). However, lists are not treated as objects, and thus cannot serve as arguments or values of functions. This limitation may be remedied in a future version of BLOG.

4.2.2 Type and function declarations

Type and function declarations do most of the work of defining a domain-specific logical language for a BLOG model (guaranteed object statements do some of this work as well). A *type declaration* has the form:

`type` τ`;`

where τ is an identifier (a string of alphanumeric characters) that will serve as the symbol for the type. A *function declaration* has one of the following forms:

`random` r f`(`$a_1, \ldots, a_k$`);`
`nonrandom` r f`(`$a_1, \ldots, a_k$`);`
`origin` r f`(`a`);`

Here f is an identifier that will serve as symbol for the function, r is a previously declared type symbol identifying the function's return type, and $a_1, \ldots, a_k$ are previously declared type symbols identifying the function's argument types. If $k = 0$

— that is, if f is a constant symbol — then the parentheses may be omitted. The keywords `random`, `nonrandom` and `origin` specify how a function's values are determined. *Random functions* are those whose values vary across possible worlds; they have their values set (on each tuple of arguments) by steps in the generative process. The values of *nonrandom functions* are fixed across all worlds. *Origin functions* play a special role in scenarios where objects generate other objects, as discussed in Sec. 4.2.6 below. Note that an origin function must take exactly one argument.

4.2.3 Guaranteed object statements

A *guaranteed object statement* asserts that certain objects are known to exist and to be distinct, and also declares constant symbols for these objects. For instance, line 5 of Figure 4.1 asserts that in every possible world, there are two distinct objects of type Color denoted by the constant symbols Blue and Green. The general form of a guaranteed object statement is:

`guaranteed` τ $c_1, \ldots, c_n$`;`

where τ is a user-defined (*i.e.*, not built-in) type symbol and $c_1, \ldots, c_n$ are identifiers that will serve as constant symbols. A BLOG model may contain multiple guaranteed object statements for the same type: all the constant symbols in all these statements denote distinct objects of the given type (it is an error to include the same constant symbol more than once). Thus, a single guaranteed object statement is not necessarily an exhaustive list of the guaranteed objects of a given type. However, the full set of guaranteed object statements in a model is exhaustive, in that each object that exists in any possible world must be introduced in some guaranteed object statement or be generated by some number statement (this will be enforced by Definition 4.10). The order in which guaranteed objects are introduced is relevant:

CPDs such as `TabularCPD` interpret their probability parameters as corresponding to guaranteed objects in they order they were declared.

BLOG also includes a shorthand syntax for defining many guaranteed objects at once. One can write a statement of the form:

`guaranteed` τ c`[`n`];`

to define n guaranteed objects of type τ, denoted by the symbols $c1, c2, c3, \ldots$. For instance, we could abbreviate line 6 in Figure 4.1 as:

```
guaranteed Draw Draw[4];
```

The guaranteed object statements for type τ in M jointly define a set of guaranteed objects $\text{Guar}_M(\tau)$. We let the constant symbols in these statements serve as their own denotations, so $\text{Guar}_M(\tau)$ is just the set of constant symbols introduced by guaranteed object statements for type τ.[1] For built-in types τ, $\text{Guar}_M(\tau)$ is the same in every model M: it is the extension given in Table 4.1. The *object set* for type τ in model M is:

$$\mathcal{O}_M(\tau) \triangleq \bigcup_{\omega \in \Omega_M} [\tau]^\omega$$

Because the set of objects of type τ can vary from world to world, $\mathcal{O}_M(\tau)$ may be a strict superset of $\text{Guar}_M(\tau)$. The *non-guaranteed objects* of type τ in model M are:

$$\text{NonGuar}_M(\tau) \triangleq \mathcal{O}_M(\tau) \setminus \text{Guar}_M(\tau)$$

We will also write NonGuar_M for the union of $\text{NonGuar}_M(\tau)$ over all types τ in $\mathcal{L}_M$.

[1] This does not preclude the possibility that some other constant symbols — introduced in function declarations rather than guaranteed object statements — might end up denoting the same objects.

4.2.4 Nonrandom function definitions

We mentioned above that a function symbol can be declared as `nonrandom`, meaning that it has the same interpretation in all possible worlds. Nonrandom function symbols serve two main purposes. First, they may represent mathematical functions that are not built into BLOG. For instance, in the aircraft tracking example, we could include a nonrandom function IsInSphere(s, r), which takes an aircraft state vector s and returns true if the position portion of that vector is in the sphere of radius r around the origin. However, nonrandom function symbols can also be used to represent known, scenario-specific information about guaranteed objects. For instance, suppose that in the urn-and-balls scenario, each draw from the urn is performed by some person; perhaps the identity of that person influences the error rate for ObsColor. If we know which person performed each draw, then there is no reason to define a prior probability model for the function Agent(d) that returns the person responsible for draw d. Instead, we can let Agent be nonrandom.

Considering the broad range of nonrandom functions that one might want to define, we do not attempt to include general syntax for specifying interpretations in BLOG. Existing programming languages are fine tools for specifying nonrandom functions. Since our BLOG engine is implemented in Java, we specify functions using instances of Java classes that implement an interface called `FunctionInterp`. A *nonrandom function definition* binds a nonrandom function symbol to a Java `FunctionInterp` object, using the syntax:

`nonrandom` f `= interp` c`[`p_1`, ..., `p_n`];`

Here f is a k-ary function that has already been declared, c is the name of a Java class that implements `FunctionInterp`, and $p_1, \ldots, p_n$ are expressions that serve as parameters to c. For example, to specify the interpretation for Agent, we could write:

```
nonrandom Agent = interp DrawToPersonInterp["agents.dat"];
```

Here we are assuming that `agents.dat` is a file containing the mapping from draws to people in some format, and `DrawToPersonInterp` is a class that can read that format and define a function from guaranteed objects of type Draw to guaranteed objects of type Person. When the BLOG engine processes this statement while loading a model, it creates an instance of class `DrawToPersonInterp`, passing the string `"agents.dat"` to that instance's constructor.

The expressions $p_1, \ldots, p_n$ that serve as parameters may be terms or formulas of $\mathcal{L}_M$ that are well-formed in the empty scope, or set expressions (see Section 4.2.8). These expressions must be *syntactically nonrandom*: that is, they must not contain any random function symbols, and must not include quantifiers or set expressions ranging over types that have number statements in M. Furthermore, it must be possible to compute the values of all the parameters without invoking the interpretation of the function f — that is, the definitions of nonrandom functions must be acyclic (the BLOG engine checks for cyclic definitions and gives error messages when they occur). If there are no parameters in a nonrandom function definition, then the square brackets after the class name can be omitted.

BLOG includes a standard `FunctionInterp` class called `ConstantInterp` that can serve as an interpretation for zero-ary function symbols (that is, constant symbols). The `ConstantInterp` class expects one parameter: a term specifying the value of the constant symbol. In fact, BLOG includes a special syntax for definitions that use `ConstantInterp`. A statement of the form:

`nonrandom` f `=` t`;`

where f is a zero-ary function and t is a syntactically nonrandom term, is an abbreviation for:

`nonrandom` f `= interp ConstantInterp[`t`];`

BLOG also allows nonrandom functions to be declared and defined in a single statement. For instance, the statement:

```
nonrandom Person Agent(Draw d)
       = interp DrawToPersonInterp["agents.dat"];
```

is both a declaration and a definition of the function Agent. A BLOG model must contain exactly one function definition statement for each nonrandom function that it declares.

We can also give a more mathematical definition of a nonrandom function interpretation that does not depend on any particular implementation language (such as Java).

Definition 4.1. *In a BLOG model M, a* nonrandom function interpretation *for a function with type signature $(r, a_1, \ldots, a_k)$ is a function from* $\mathrm{Guar}_M(a_1) \times cdots \times \mathrm{Guar}_M(a_k)$ *to* $\mathrm{Guar}_M(r) \cup \{\mathsf{null}\}$.

We will use $[f]_M$ to denote the interpretation that M assigns to a nonrandom function f.

4.2.5 Dependency statements

Dependency statements specify probability distributions for the values of random functions. As an example, consider the dependency statement for $\mathsf{State}(a, t)$ in Figure 4.4:

```
State(a, t)
    if t = 0 then ~ InitState()
    else ~ StateTransition(State(a, Pred(t)));
```

In the generative process, this statement is applied for every Aircraft object a and every natural number t. If $t=0$, the conditional distribution for $\mathsf{State}(a,t)$ is given by the *elementary CPD* `InitState`; otherwise it is given by the elementary CPD `StateTransition`, which takes $\mathsf{State}(\mathsf{a},\mathsf{Pred}(\mathsf{t}))$ as an argument. These elementary CPDs define distributions over objects of type R6Vector (the return type of State). In our implementation, elementary CPDs are instances of Java classes that implement an interface called `CondProbDistrib`.

A BLOG model contains exactly one dependency statement for each random function (the order of these statements is irrelevant). The general syntax for a dependency statement is as follows:

$f(x_1, \ldots, x_k)$
    `if` φ_1 `then` $\sim$ c_1`(`e_{11}`, ...,` e_{1n_1}`)`
    `elseif` φ_2 `then` $\sim$ c_2`(`e_{21}`, ...,` e_{2n_2}`)`
    $\vdots$
    `else` $\sim$ c_m`(`e_{m1}`, ...,` e_{mn_m}`);`

The statement begins with a header, which includes the random function symbol f and logical variables $x_1, \ldots, x_k$ that stand for f's arguments. This function f is called the *child function* in the dependency statement (by analogy to the child variable for a CPD in a Bayesian network). The header defines a scope $\beta_f = \{(x_1, a_1), \ldots, (x_k, a_k)\}$, where $a_1, \ldots, a_k$ are the types of f's arguments.

The remainder of the statement defines a sequence of *clauses*. A clause consists of a condition φ, an elementary CPD c, and a tuple of expressions $(e_1, \ldots, e_n)$ that serve as arguments to the CPD. The syntactic representation of a clause begins with `if` for the first clause, `elseif` for subsequent clauses, and `else` for the last clause: as these keywords suggest, the clause that is active in a particular world is the first clause whose condition is satisfied. The condition in a clause can be any formula

of $\mathcal{L}_M$ that is well-formed in the scope β_f. The condition for the last clause is not specified explicitly: it is always simply true. If desired, the set of clauses (that is, the portion of the dependency statement between the header and the final semicolon) can be enclosed in curly braces for clarity.

After the condition, each clause specifies an elementary CPD for the range $\mathcal{O}_M(\text{ret}f) \cup \{\text{null}\}$. This specification can consist of just a Java class name, such as `StateTransition`, or it may consist of a Java class name followed by some parameters in square brackets. These parameters are not to be confused with the CPD arguments that come after the elementary CPD; for instance, in:

```
TabularCPD[[0.8, 0.2], [0.2, 0.8]](TrueColor(BallDrawn(d)))
```

the vectors $[0.8, 0.2]$ and $[0.2, 0.8]$ are parameters, whereas TrueColor(BallDrawn(d)) is an argument. The parameters must be nonrandom, and cannot depend on the function arguments $x_1, \ldots, x_k$. Formally, the parameters may be any syntactically nonrandom terms, formulas, or set expressions that are well-formed in the empty scope. When a dependency statement is processed by the BLOG engine, the parameters of each elementary CPD are evaluated, and their values are passed to the constructor of the given Java class to create a new instance. For example, if a model includes two elementary CPD specifications `Poisson[6]` and `Poisson[8]`, then the engine creates two instances of the Java class `Poisson`, one with mean 6 and one with mean 8. These Java objects define elementary CPDs, in a sense that we make precise in Section 4.2.9.

Finally, a clause specifies a tuple of CPD arguments $e_1, \ldots, e_n$. Unlike CPD parameters, which are evaluated at initialization time without reference to any particular world, CPD arguments take on values that depend on the possible world and the assignment of values to the variables in β_f. The arguments can be any terms,

formulas, or set expressions that are well-defined in β_f. Examples of arguments include {Ball b} and Name(Author(PubCited(c))) in Figure 4.1, and State(a, t) in Figure 4.4.

We have given the syntax for fully general dependency statements, but our example models rarely use this full-fledged if-then-else form: they use certain abbreviations. The allowed abbreviations are as follows.

- If we want to define a dependency model with just a single clause whose condition is `true`, we can dispense with the `if` and `then` keywords and write a statement of the form:

  f`(`x_1`, ..., `x_k`) ~ `c`(`e_1`, ..., `e_n`);`

  Line 12 of Figure 4.2 is a good example of this format.

- There is a special `CondProbDistrib` class called `EqualsCPD` that is used to represent deterministic dependencies. `EqualsCPD` takes a single expression e as an argument, and constrains the child function to take the value of that expression with probability one. The expression e may be a term, in which case the return type of the child function f must be the same as the type of the term; a formula, in which case ret(f) must be Boolean; or a cardinality expression (see Section 4.2.8 below), in which case ret(f) must be NaturalNum. Instead of writing ~ `EqualsCPD(`e`)`, we can write `=` e as a shorthand. For example, in a version of the urn-and-balls scenario with deterministic observations, we could write:

  ```
  ObsColor(d) = TrueColor(BallDrawn(d));
  ```

- If a dependency statement contains one or more clauses with the `if` and `elseif` keywords but no `else` clause, then there is an implicit final clause of the form:

`else =` *default*

where *default* is false when the child function is Boolean, and null when the child function has any other return type.

- If the child function has no arguments, then we can omit the empty parentheses. We see this in the dependency statement for the random constant symbol First in Figure 4.2. Similarly, if an elementary CPD takes no arguments, the empty parentheses can be omitted after it.

Finally, just as we can combine function declarations with nonrandom function definitions, BLOG also allows us to combine function declarations with dependency statements. We simply add the `random` keyword and type information to the header. So, for example, we could combine lines 3 and 9 in Figure 4.1 to yield:

```
random Ball BallDrawn(Draw d) ~ Uniform({Ball b});
```

4.2.6 Number statements

Number statements specify how objects are added to the world in the generative process described by a BLOG model. In the simple case, a model contains at most one number statement for each type: then each number statement defines a distribution for the total number of non-guaranteed objects of that type. However, as we saw in the aircraft tracking example (Figure 4.4), it is not always convenient to assume that all the non-guaranteed objects of a given type are added to the world in a single generative step. The radar blips in this example are generated in many separate steps: one for each aircraft at each time point, plus another step at each time point to generate "false alarm" blips. We can think of blips as being generated by other objects — namely aircraft and natural numbers (time points) — and being tied back to those generating objects by the origin functions Source and Time. Line

10 in Figure 4.4 also specifies that the number of blips generated by aircraft a at time t depends on $\mathsf{State}(a, t)$.

Thus, the general syntax of a number statement is as follows:

```
#τ(g_1 = x_1, ..., g_k = x_k)
    if φ_1 then ~ c_1(e_11, ..., e_1n_1)
    elseif φ_2 then ~ e_2(e_21, ..., e_2n_2)
    ⋮
    else ~ c_m(e_m1, ..., e_mn_m);
```

This is the same syntax as for dependency statements, except for the header. Here, the header consists of a user-defined type symbol τ, a sequence of distinct origin function symbols $g_1, \ldots g_k$ that take an argument of type τ, and a sequence of logical variables $x_1, \ldots, x_k$ that stand for the generating objects, *i.e.*, the return values of the origin functions. The header defines a scope $\{(x_1, r_1), \ldots, (x_k, r_k)\}$, where $r_1, \ldots, r_k$ are the origin functions' return types; expressions in the rest of the number statement are evaluated in this scope. As usual, if a number statement involves no origin functions, the empty parentheses can be omitted.

The header for a number statement does not need to include all the origin functions for a given type: for example, line 11 in Figure 4.4 does not include the function Source. When an origin function is not included, its value on the generated objects defaults to null. A BLOG model can contain any number of number statements for a given type, as long as no two number statements use the same set of origin functions. This restriction ensures that one can always tell which number statement generated a non-guaranteed object o by looking at the values of the origin functions on o.

After the header, a number statement defines a sequence of clauses, just as in a dependency statement. The abbreviations that apply to dependency statements (omitting "`if`...`then`" when the condition is just `true`, the implicit `else` clause, etc.)

can also be used in number statements. The only differences are that all the elementary CPDs used in a number statement must have $\mathbb{N}$ as their range set, and the default value specified by the implicit `else` clause is zero rather than false or null.

Note that the number of objects added in any single generative step is always finite. However, object generation can be recursive: objects can generate other objects of the same type. For instance, consider a model of sexual reproduction in which every male–female pair of individuals produces some number of offspring. We could represent this with the number statement:

```
#Individual(Mother = m, Father = f)
        if Female(m) & !Female(f) then ~ NumOffspringPrior;
```

If a model contains recursive number statements, the total number of non-guaranteed objects in a possible world may be infinite, even though each generative step adds a finite number of them.

Another way to obtain an infinite set of non-guaranteed objects is by letting them be generated by the natural numbers. For instance, to model a version of the urn-and-balls scenario with infinitely many draws, we could use the following statements instead of the guaranteed object statement for draws:

```
origin NaturalNum Index(Draw); #Draw(Index = n) = 1;
```

These statements ensure that for each natural number n, there is exactly one draw whose index is n (recall that "`= 1`" is an abbreviation for "`~ EqualsCPD(1)`"). However, the statements do not give us any constant symbols, or even any terms, that we can use to refer to draws. We can remedy this with the following statements:

```
random Draw NthDraw(NaturalNum);
NthDraw(n) ~ Iota({Draw d:  Index(d) = n});
```

Logical syntax	BLOG syntax
$t_1 = t_2$	t_1 `=` t_2
$t_1 \neq t_2$	t_1 `!=` t_2
$\neg\psi$	`!`ψ
$\psi \wedge \chi$	ψ `&` χ
$\psi \vee \chi$	ψ `|` χ
$\psi \rightarrow \chi$	ψ `->` χ
$\forall\ \tau\ v\ \psi$	`forall` $\tau\ v\ \psi$
$\exists\ \tau\ v\ \psi$	`exists` $\tau\ v\ \psi$

Table 4.3: Standard syntax and BLOG syntax for logical formulas.

Here `Iota` is an elementary CPD that takes in a set of size one, and defines a distribution that puts probability one on the sole element of that set (if the given set is not of size one, it puts probability one on null). Once these statements are included, we can refer to draws using terms such as NthDraw(14). This is admittedly a rather complicated solution for a simple modeling task; future versions of BLOG may include syntax to make this more straightforward.

4.2.7 Logical formulas

We have said in a BLOG model M, the formulas of the logical language $\mathcal{L}_M$ can serve as constituents in nonrandom function definitions, dependency statements, and number statements. However, when we defined first-order logical languages in Section 2.1.3, we used non-ASCII characters such as $\neg$, $\wedge$, and $\forall$. Table 4.3 shows how we replace such characters with ASCII equivalents.

4.2.8 Set expressions

In this section, we give an overview of the set expressions that BLOG supports, followed by a formal definition of their syntax and semantics. As we shall see, the term "set expression" is something of a misnomer: while some of these expressions do

denote sets, others denote multisets, and one kind of set expression denotes a number that is the cardinality of a set. However, "set expression" remains a convenient way of referring to these various expressions that use set notation.

We have seen several places in our examples where a set expression is used as an argument to an elementary CPD. The first of these is on line 9 of Figure 4.1, where we pass {`Ball b`} into the `Uniform` CPD. In an alternative version of the urn-and-balls model where there are multiple urns, we could use a set expression such as {`Ball b : In(b, Urn1)`}. In these cases, the goal is to define a distribution over a set of objects that varies from world to world; the CPD needs to take this set as an argument so that it can define the desired distribution.

The `Uniform` CPD selects uniformly from a given set, but what if we want a CPD to do weighted sampling? For example, we might want to model a scenario where the chance that a ball is selected on any given draw depends on its color. In this case, we could use an elementary CPD `SampleGivenColor` that takes not just a set of balls, but a set of pairs (b, c) where b is a ball and c is its color. The BLOG syntax for this is:

```
{b, TrueColor(b) for Ball b}
```

We call this kind of expression a *tuple multiset expression*; it defines a multiset of tuples. A *multiset* U consists of a set set_U and a function $\text{mult}_U : \text{set}_U \to \mathbb{N}$ that returns a *multiplicity* for each element of set_U, representing the number of times that element occurs. In our example, the multiplicity of each tuple is one, because each value of the logical variable b yields a different tuple. But if the expression were simply {`TrueColor(b) for Ball b`}, several values of b could yield the same tuple (here each tuple has just one element, which is a color). If U is a multiset, we will simply write $o \in U$ to mean $o \in \text{set}_U$. Also, if S is a set, we will use the notation $\{|f(x) : x \in S|\}$ to represent the multiset U with $\text{set}_U = \{f(x) : x \in S\}$

and $\text{mult}_U(o) = |\{x \in S \,:\, f(x) = o\}|$. We will use the same notation to define one multiset in terms of another one: if U is a multiset, then $\{|f(x) \,:\, x \in U|\}$ is the multiset U' with $\text{set}_{U'} = \{f(x) \,:\, x \in U\}$ and $\text{mult}_{U'}(o) = \sum_{(x \in U \,:\, f(x) = o)} \text{mult}_U(x)$.

Set expressions are also useful when a conditional distribution depends on the attributes of a set of objects. For instance, lines 17–20 in Figure 4.3 say that the text of a citation c depends on the names of all the authors of PubCited(c). The tuple multiset expression used here is:

```
{n, Name(NthAuthor(PubCited(c), n))
                    for NaturalNum n : n < NumAuthors(PubCited(c))}
```

This expression evaluates to a multiset of pairs (n, s) where n is a natural number and s is a string (an author name); these pairs help determine the distribution over citation strings. The numbers need to be included in these pairs so that the distribution over citation strings reflects the order of the authors (a future version of BLOG may include a new kind of set expression that evaluates to a list rather than a multiset). CPDs with tuple multisets as arguments can also perform more standard aggregation operations, such as taking an average or median of a set of real numbers. For instance, suppose the chance of a paper being accepted to a conference depends on the average intelligence of its authors. If Intelligence is a function from Researcher to Real, then we can write the dependency statement:

```
Accepted(p) ~ AcceptanceCPD({Intelligence(NthAuthor(p, n))
                                for NaturalNum n : n < NumAuthors(p)});
```

This `AcceptanceCPD` could, for example, take the average of the real numbers passed into it, and pass this average through a logistic function to get the probability of acceptance. It might be useful in future version of BLOG to allow aggregation functions

(such as Average) to be separated from CPDs, so elementary CPDs and aggregation functions could be mixed and matched in a compositional way. Currently, however, BLOG has no facility for defining functions on sets or multisets.

There are two other kinds of set expressions that BLOG supports. One is an *explicit set expression*, in which the elements of the set are listed explicitly. For example, in the hurricane model (Figure 4.2), we could use the explicit set expression `{A, B}` rather than the implicit set expression `{City c}` on line 8. The last kind of set expression is a *cardinality expression*, which yields the size of an implicitly defined set. In the urn-and-balls example, we could use the cardinality expression `#{Ball b: TrueColor(b) = Blue}` to represent the number of blue balls in the urn.

We now formally define the syntax of BLOG set expressions. This definition uses the notion of a *scope* (a mapping from logical variables to types) from Section 2.1.3.

Definition 4.2. *A* well-formed set expression *of a BLOG model M in a scope β has one of the following forms:*

- *an* explicit set expression $\{t_1\texttt{, }\ldots\texttt{, }t_k\}$*, where $t_1, \ldots, t_k$ are well-formed terms of $\mathcal{L}_M$ in scope β;*
- *an* implicit set expression $\{\tau\ x\ \texttt{:}\ \varphi\}$*, where τ is a type in M, x is a logical variable symbol, and φ is a formula of $\mathcal{L}_M$ that is well-formed in the scope $(\beta; x \mapsto \tau)$;*
- *a* tuple multiset expression $\{t_1\texttt{, }\ldots\texttt{, }t_k\ \texttt{for}\ \tau_1\ x_1\texttt{, }\ldots\texttt{, }\tau_n\ x_n\ \texttt{:}\ \varphi\}$*, where $\tau_1, \ldots, \tau_n$ are types in M, $x_1, \ldots, x_n$ are logical variable symbols, $t_1, \ldots, t_k$ are terms of $\mathcal{L}_M$ that are well-formed in the scope $\beta' \equiv (\beta; x_1 \mapsto \tau_1, \ldots, x_k \mapsto \tau_n)$, and φ is a formula of $\mathcal{L}_M$ that is well-formed in β';*
- *a* cardinality expression $\#s$*, where s is a well-formed implicit set expression of M in β.*

If the condition φ in an implicit set expression or tuple multiset expression is simply true, then the condition and the colon that precedes it can be omitted. {`Ball b`} is an example of such an abbreviated expression.

A set expression may represent an infinite set, such as {`NaturalNum n`}, or {`Individual i : Female(i)`} in the recursive sexual reproduction model that we mentioned near the end of Section 4.2.6. In our Java implementation of BLOG, we do not have a general way to represent infinite sets or multisets so that they can be passed as arguments to elementary CPDs. Furthermore, since the number of infinite subsets of a given universe is uncountable, set expressions that can denote infinite sets cannot be thought of as discrete random variables. Thus, the elementary notion of conditional probability that we use throughout this thesis (see Section 2.2.6) does not allow us to define a CPD for a random variable given the value of such a set expression. To avoid these difficulties, we adopt the convention that a set expression that would otherwise denote an infinite set or multiset actually denotes null. Similarly, a cardinality expression involving an infinite set also denotes null.

Just as we defined the denotation of a term given a model structure and an assignment of values to logical variables (Definition 2.7), we can make a similar definition for set expressions.

Definition 4.3. *Let ω be a model structure of $\mathcal{L}_M$, α be an assignment that is valid in ω, and s be a set expression of M that is well-formed in* $\mathrm{domain}(\alpha)$*. Then the* denotation *of s in ω under α, denoted $[s]^\omega_\alpha$, is defined as follows:*

- *if s is an explicit set expression $\{t_1, \ldots, t_k\}$, then $[s]^\omega_\alpha$ is the set $\{[t_i]^\omega_\alpha : i \in \{1, \ldots, k\}\}$;*

- *if s is an implicit set expression $\{\tau\ x : \varphi\}$, then $[s]^\omega_\alpha$ is the set $\{o \in [\tau]^\omega : \omega \models_{(\alpha,(x,\tau)\mapsto o)} \varphi\}$, or* null *if that set is infinite;*

- *if s is a tuple multiset expression* $\{t_1, \ldots, t_k \texttt{ for } \tau_1\ x_1, \ldots, \tau_n\ x_n : \varphi\}$ *and we define A to be the set of valid assignments* $\alpha' = (\alpha; (x_1, \tau_1) \mapsto o_1, \ldots, (x_n, \tau_n) \mapsto o_n)$ *in ω such that $\omega \models_{\alpha'} \varphi$, then* $[s]^\omega_\alpha$ *is the multiset* $\{|([t_1]^\omega_{\alpha'}, \ldots, [t_k]^\omega_{\alpha'}) : \alpha' \in A|\}$, *or* null *if A is infinite;*

- *if s is a cardinality expression* $\#s'$, *then* $[s]^\omega_\alpha$ *is* $|\,[s']^\omega_\alpha\,|$, *or* null *if* $[s']^\omega_\alpha = \mathsf{null}$.

Thus set expressions, like terms and formulas, can be evaluated to yield values in a possible world. The values of set expressions are always finite sets, finite multisets, natural numbers (for cardinality expressions), or null. From here on, we will refer to terms, formulas, and set expressions collectively as *BLOG expressions*. Also, for consistency, we will sometimes use the notation $[\varphi]^\omega_\alpha$ for the value of a formula φ in a model structure ω. That is, $[\varphi]^\omega_\alpha$ equals true if $\omega \models_\alpha \varphi$, and equals false otherwise.

4.2.9 Elementary CPDs

We have said so far that an elementary CPD is an instance of a Java class that implements a certain interface. But we would like our definition of a BLOG model to be independent of any particular implementation language. Thus, we give a more mathematical definition of an elementary CPD below. This definition enforces two assumptions, which can be stated informally as follows. If an elementary CPD is used in a model M and invoked with argument values $q_1, \ldots, q_n$:

1. the CPD can assign positive probability to an object o only if o is a guaranteed object in M or is contained in the argument tuple $(q_1, \ldots, q_n)$;

2. if h is a permutation of the non-guaranteed objects in M, then the probability that the CPD assigns to a non-guaranteed object o given $q_1, \ldots, q_n$ is the same as it assigns to $h(o)$ given $h(q_1), \ldots, h(q_n)$.

The first condition ensures that the CPD does not assign positive probability to objects that do not exist. The second condition asserts that non-guaranteed objects — for example, the balls in the urn-and-balls example — are treated interchangeably. Note that elementary CPDs are allowed to make distinctions among guaranteed objects: for instance, the tabular CPD for Damage in Figure 4.2 distinguishes between the two guaranteed PrepLevel objects that might be passed into it.

The way we stated these assumptions above ignores the fact that the arguments passed into an elementary CPD are not necessarily individual objects; they may be sets of objects, or multisets of tuples of objects, defined by a set expression. Thus, we need to be more careful in saying what it means for an expression value to "contain" an object, and what it means to apply a permutation h to an expression value.

Definition 4.4. *In a BLOG model M, an expression value q* contains *an object o if either $q = o$, or $q \notin \text{NonGuar}_M$ and one of the following conditions holds:*

- *q is a set and for some $q' \in q$, q' contains o;*
- *q is a multiset and for some $q' \in q$, q' contains o;*
- *q is a tuple $(q_1, \ldots, q_k)$ and for some $i \in \{1, \ldots, k\}$, q_i contains o.*

Note that the recursion in this definition stops when it hits a non-guaranteed object, even if that non-guaranteed object happens to be a tuple (we will see in Section 4.3 that it is convenient to let non-guaranteed objects be tuples).

Definition 4.5. *Let M be a BLOG model, and let h be a permutation of* NonGuar_M*: that is, a bijection from* NonGuar_M *to itself. The* extension of h to argument values *in M, denoted $\bar{h}$, is defined as follows on any expression value q:*

- *if $q \in \text{NonGuar}_M$, then $\bar{h}(q) = h(q)$;*
- *otherwise:*

- *if q is a set, then* $\bar{h}(q) = \{\bar{h}(q') : q' \in q\}$;
- *if q is a multiset, then* $\bar{h}(q) = \{|\bar{h}(q') : q' \in q|\}$;
- *if q is a tuple* $(q_1, \ldots, q_k)$, *then* $\bar{h}(q) = (\bar{h}(q_1), \ldots, \bar{h}(q_k))$;
- *otherwise,* $\bar{h}(q) = q$.

We can now state a formal definition of an elementary CPD, including the restrictions that we stated less formally at the beginning of this section.

Definition 4.6. *An* elementary CPD *for a range set S in a discrete BLOG model M is a function c from the set of pairs $(o, (q_1, \ldots, q_k))$, where $o \in S$ and $(q_1, \ldots, q_k)$ is any tuple of expression values, to $[0, 1]$, such that:*

- *for each tuple of expression values* $(q_1, \ldots, q_k)$, $\sum_{(o \in S)} c(o, (q_1, \ldots, q_k)) = 1$;
- *if* $c(o, (q_1, \ldots, q_k)) > 0$, *then* $o \in (\mathrm{Guar}_M(\tau) \cup \{\mathsf{null}\})$ *or* q_i *contains* o *for some* $i \in \{1, \ldots, k\}$;
- *for any permutation h on* $\mathrm{NonGuar}_M$,

$$c(o, (q_1, \ldots, q_k)) = c(\bar{h}(o), (\bar{h}(q_1), \ldots, \bar{h}(q_k)))$$

for any object o and expression values $(q_1, \ldots, q_k)$.

Intuitively, the value $c(o, (q_1, \ldots, q_k))$ is to be interpreted as the conditional probability of the value o given the CPD arguments $q_1, \ldots, q_k$. As specified in previous sections, elementary CPDs in the dependency statement for a random function with return type r must have $\mathcal{O}_M(r) \cup \{\mathsf{null}\}$ as their range set; the range set for elementary CPDs in number statements must be $\mathbb{N}$. Section 4.3.4 describes more formally the role that elementary CPDs play in the semantics of BLOG. In the full version of BLOG that includes Real and RkVector types, $c(o, (q_1, \ldots, q_k))$ is to be interpreted

as a density at o rather than a probability; in this case, the value can range over $[0, \infty)$, not just $[0, 1]$.

Definition 4.6 requires an elementary CPD to accept any tuple of expression values $(q_1, \ldots, q_k)$ as arguments. In practice, an instance of the `TabularCPD` class expects a certain number of arguments, each being a single object; an instance of class `Uniform`, on the other hand, expects a single set as an argument. Currently, our interface for elementary CPDs does not include any facility for checking, when the model is loaded, that an argument list matches the CPD's expectations. During inference, if an elementary CPD gets an argument that it is not expecting, it can just assign probability 1 to `null`, and probability zero to all other values.

4.3 Declarative semantics

Generative processes provide an intuitive way to understand the semantics of BLOG models. However, we run into some difficulties if we try to formalize this generative process intuition. In what order can the generative steps be executed? Could the distribution over possible worlds depend on the execution order? Even if we did specify the generative process precisely enough for its semantics to be well-defined, such a procedural semantics might be unwieldy for deriving learning and inference algorithms and proving their correctness.

Thus, we provide a declarative semantics for discrete BLOG models, based on the partition-based models (PBMs) of Chapter 3. We will describe how a BLOG model M defines a set of possible worlds Ω_M, a set of *basic random variables* $\mathcal{V}_M$, and a PBM Γ_M over these variables. A PBM is a declarative representation: it specifies conditional probabilities for each random variable given a certain class of events, along with a set of conditional independence properties. The basic result from Chapter 3 (Theorem 3.13) is that if a PBM satisfies certain conditions, then there

is exactly one probability measure over possible worlds in which these conditional probabilities and independence properties hold. This is the probability measure that the PBM represents.

4.3.1 Event spaces for BLOG models

The set of possible worlds Ω_M of a BLOG model M consists of model structures of the logical language $\mathcal{L}_M$. Even before defining Ω_M precisely (which we do in Section 4.3.2.3), we can consider how we will define an event space on Ω_M. Note that even in discrete BLOG, Ω_M will often be uncountable: for instance, if M includes a random function on the natural numbers, then that function has uncountably many possible interpretations.

A natural way to define an event space on a set of model structures is to use the σ-field (see Section 2.2.1) generated by events that assign a value to a function on a particular tuple of arguments, or assert the existence of a particular object. In keeping with our assumptions from Chapter 3, we assume that functions take values in discrete spaces. That is, we assume that for each type τ in $\mathcal{L}_M$, $\mathcal{O}_M(\tau)$ is countable.

Definition 4.7. *The event space $\mathscr{F}_M$ for a discrete BLOG model M is the σ-field generated by the following events:*

- *for each function symbol f in $\mathcal{L}_M$ with type signature $(r, a_1, \ldots, a_k)$, each argument tuple $(o_1, \ldots, o_k) \in \mathcal{O}_M(a_1) \times \cdots \times \mathcal{O}_M(a_k)$, and each value $o \in \mathcal{O}_M(r) \cup \{\mathsf{null}\}$, the event:*

$$\{\omega \in \Omega_M \,:\, o_1 \in [a_1]^\omega, \ldots, o_k \in [a_k]^\omega \;\; \textit{and} \;\; [f]^\omega(o_1, \ldots, o_k) = o\};$$

- *for each type τ in $\mathcal{L}_M$ and each object $o \in \mathcal{O}_M(\tau)$, the event:*

$$\{\omega \in \Omega_M \,:\, o \in [\tau]^\omega\}.$$

In Appendix 4.A, we show that all expressions (terms, formulas, and set expressions) in a discrete BLOG model can be seen as random variables on this event space: the event that an expression takes on any given value is measurable. We could extend Definition 4.7 to handle real-valued functions f by replacing $[f]^\omega(o_1, \ldots, o_k) = o$ with $[f]^\omega(o_1, \ldots, o_k) \in A$, where A ranges over the Borel subsets of $\mathbb{R}$.

4.3.2 Possible worlds and basic random variables

4.3.2.1 Models with fixed object sets

If a BLOG model M contains no number statements, then the only objects that can exist in possible worlds are guaranteed objects. In this case, defining the semantics of the model is straightforward. In all possible worlds $\omega \in \Omega_M$, $[\tau]^\omega = \mathrm{Guar}_M(\tau)$ for each type τ. The set of possible interpretations for a random function symbol f with type signature $(r, a_1, \ldots, a_k)$ is the set of functions from $\mathrm{Guar}_M(a_1) \times \cdots \times \mathrm{Guar}_M(a_k)$ to $\mathrm{Guar}_M(r)$. The outcome space Ω_M contains one possible world for each way of assigning an interpretation to each random function.

Now for each random function f with type signature $(r, a_1, \ldots, a_k)$, and each tuple of arguments $o_1, \ldots, o_k \in \mathrm{Guar}_M(a_1) \times \cdots \times \mathrm{Guar}_M(a_k)$, we can define a *function application variable*:

$$V_f[o_1, \ldots, o_k](\omega) \triangleq [f]^\omega(o_1, \ldots, o_k)$$

It is easy to see that the function application variables generate $\mathscr{F}_M$. Furthermore,

every partial or complete instantiation on these basic random variables is achievable (*i.e.*, corresponds to a possible world). So the set of function application variables is sufficient for $(\Omega_M, \mathscr{F}_M)$ (Definition 3.2), and any PBM over these random variables respects its outcome space (Definition 3.12). The dependency statements of a BLOG model define a partition and CPD for each function application variable, as we will discuss below.

4.3.2.2 Models with at most one number variable per type

To increase our ambitions gradually, let us now focus on models which have unknown objects, but in which all the objects of each type are generated in a single step. These are models where the number statements do not include any origin functions, such as our models for the urn and balls and the citation scenario. We now need a more diverse set of possible worlds: the worlds differ in what objects exist, not just in the interpretations of function symbols. If the model includes a number statement for a type τ, then in each possible world, the extension of τ will include some set of non-guaranteed objects. Let us assume that these non-guaranteed objects are pairs $(\tau, 1), (\tau, 2), (\tau, 3)$, etc. For instance, in the citation model, (Publication, 7) is a non-guaranteed object of type Publication.

The function application variables are now defined not just for argument tuples consisting of guaranteed objects, but also for tuples that include non-guaranteed objects. On worlds where some of the arguments $o_1, \ldots, o_k$ do not exist, a function application variable $V_f\,[o_1, \ldots, o_k]$ yields the value null. To complete the set of basic random variables for this kind of model, we add a *number variable* N_τ for each type τ that has a number statement:

$$N_\tau(\omega) = |\,[\tau]^\omega \setminus \mathrm{Guar}_M(\tau)|$$

Note that we end up with one basic RV for each function-value-setting or object-generating step that could take place in a run of this model's generative process. CPDs for these basic RVs are specified by the dependency and number statements.

To make these basic RVs sufficient for the set of possible worlds, we need to impose a restriction: Ω_M includes only those model structures ω where for each type τ, $\{n : (\tau, n) \in [\tau]^\omega\} = \{1, 2, \ldots, n^*\}$ for some natural number n^*. Thus, for natural numbers $n \geq 1$, the event $\{\omega \in \Omega_M : (\tau, n) \in [\tau]^\omega\}$ is identical to the event $\{\omega \in \Omega_M : N_\tau(\omega) \geq n\}$. From a generative standpoint, we are saying that when a step adds n^* objects to the world, these objects are pairs $(\tau, 1), (\tau, 2), \ldots, (\tau, n^*)$, not some arbitrary set of pairs.

4.3.2.3 Models with multiple number variables per type

As we noted in the aircraft tracking example, it is sometimes convenient to think of the objects of a particular type (such as Blip) as being generated in many separate steps, rather than all at once. In such models, some types have multiple number variables.

Let us put aside generative processes for a moment and just consider PBMs over sets of variables that include both function application variables and number variables — where by "number variables" we mean random variables that return the number of objects having a particular property in a given world. The properties that the number variables refer to cannot be chosen arbitrarily. For instance, in the citation example (Figure 4.3), suppose we introduce number variables $N_{\mathsf{Publication}}[o]$ for each researcher o, returning the number of publications q that have $[\mathsf{NthAuthor}]^\omega(q, n) = o$ for any number n. If we still include function application variables for NthAuthor, then these function application variables are highly constrained by the $N_{\mathsf{Publication}}[o]$ variables. For example, in a world where $N_{\mathsf{Publication}}[o] = 4$, there must be exactly four variables of the form $V_{\mathsf{NthAuthor}}[q, n]$ that have o as their value.

In order to respect its outcome space, a PBM would need to define tight dependencies between such variables. On the other hand, if we omit the function application variables for NthAuthor, we get an underspecified model that is not sufficient for the outcome space. Such a model does not specify the chance that two given researchers end up as authors of the same publication.

The system of origin functions and number statements in BLOG is crafted to avoid such difficulties. For each number statement for type τ with origin functions $g_1, \ldots, g_k$, and each tuple of appropriately typed generating objects $o_1, \ldots, o_k$, there is a number variable $N_\tau\,[g_1 = o_1, \ldots, g_k = o_k]$. The value of this number variable in a world ω is the number of non-guaranteed objects that satisfy this number statement applied to these generating objects, in the following sense.

Definition 4.8. *Consider a number statement for type τ with origin functions $g_1, \ldots, g_k$. In a model structure ω, an object $q \in [\tau]^\omega$* satisfies *this number statement applied to $o_1, \ldots, o_k$ in ω if $[g_i]^\omega(q) = o_i$ for $i = 1, \ldots, k$, and $[g]^\omega(q) = \mathsf{null}$ for all other τ-origin functions g.*

Note that this definition implies that a given object can satisfy at most one number statement applied to at most one tuple of generating objects in a given world. As a shorthand, we will sometimes say that an object satisfies a number variable $N_\tau\,[g_1 = o_1, \ldots, g_k = o_k]$, when we mean that it satisfies that variable's underlying number statement applied to the generating objects $o_1, \ldots, o_k$. Because the sets of objects that satisfy distinct number variables are necessarily disjoint, we do not have to worry about defining the probability that an object is in two such sets. And to avoid conflicts with function application variables for the origin functions, we simply omit those function application variables from our basic variable set.

However, there is still an issue to resolve: although the number variables for type τ jointly specify how many objects of type τ exist in a given world, they do

not directly specify which objects satisfy which number statement applications. For instance, the number variables might specify that there is one blip whose source is $(\mathsf{Aircraft}, 20)$ and whose time is 8, but no blips whose source is $(\mathsf{Aicraft}, 21)$ and whose time is 8. But given such an instantiation, there is still ambiguity about *which* blip — out of the infinite number of blips that may exist in a possible world — has source $(\mathsf{Aircraft}, 20)$ and time stamp 8. This is a symptom of an underspecification in our generative process: we have not specified which objects are added by which generative steps (only how many are added by each step).

Thus, we need to further constrain the outcome space to allow these basic random variables to generate the event space $\mathscr{F}_M$ (particularly those events that specify the values of origin functions on objects). There are two paths we could take to do this. The first is to define an ordering on the number variables for each type. The ordering could be based on the sum of the indices of the generating objects: thus the ordering on aircraft-time pairs would begin $((\mathsf{Aircraft}, 0), 0), ((\mathsf{Aircraft}, 0), 1), ((\mathsf{Aircraft}, 1), 0)$, etc. Then non-guaranteed objects could be assigned to number variables in numerical order. For example, in each possible world, $(\mathsf{Blip}, 0)$ would be assigned to the first number variable with a nonzero value; $(\mathsf{Blip}, 1)$ would satisfy the same number variable if its value was greater than 1, and the next nonzero number variable otherwise; and so on.

This approach can be made to work, in the sense of ensuring that the basic RVs defined above are sufficient for the set of possible worlds. But under this approach, determining the value of an origin variable on an object (τ, n) in a given world involves iterating over all the number variables in order until the sum of their values reaches n. And we need to find the values of origin variables in order to identify the dependencies between basic RVs: for instance, $\mathsf{MeasuredPos}(b)$ depends on $\mathsf{State}(\mathsf{Source}(b), \mathsf{Time}(b))$. Thus, the current approach leads to two problems: finding the values of origin functions is time-consuming, and variables such as

$\mathsf{MeasuredPos}(b)$ end up depending probabilistically on all the number variables that could be relevant in determining what $\mathsf{Source}(b)$ and $\mathsf{Time}(b)$ are. These are problems for inference algorithms, not for the semantics of the model. However, since we will be developing inference algorithms in the next chapter, we would like to avoid imposing these problems on ourselves if possible.

In fact, there is a second way to resolve the ambiguity about which objects satisfy which number variables. The trick is to change the representation of non-guaranteed objects so that each object lists its own generating objects. Specifically, in a world where a number variable $N_\tau\left[g_1 = o_1, \ldots, g_k = o_k\right]$ has the value n^*, the objects that satisfy it will be tuples $(\tau, (g_1, o_1), \ldots, (g_k, o_k), n)$ for $n \in \{1, \ldots, n^*\}$. This representation boils down to our earlier pair representation for objects with no generating objects: thus aircraft are still represented as $(\mathsf{Aircraft}, 0)$, $(\mathsf{Aircraft}, 1)$, etc. But a single blip generated by $(\mathsf{Aircraft}, 20)$ at time 8 is represented as:

$$(\mathsf{Blip}, (\mathsf{Source}, (\mathsf{Aircraft}, 20)), (\mathsf{Time}, 8), 0)$$

Thus, the representations for non-guaranteed objects end up being *nested* tuples that encode their whole generation history. Determining the values of origin functions on an object is no longer an expensive task, and no longer introduces unnecesssary probabilistic dependencies into the model.

The following definitions formalize this approach to defining possible worlds and basic random variables. First, we define the *universe* for each type: the set of objects that may be in the extension of that type in some possible world.

Definition 4.9. *In a BLOG model M, the sets $\mathcal{U}_M^i(\tau)$ of* level–i *objects for each*

type τ are defined inductively for natural numbers i, as follows:

$$\begin{aligned}
\mathcal{U}_M^0(\tau) &= \mathrm{Guar}_M(\tau) \\
\mathcal{U}_M^{i+1}(\tau) &= \mathcal{U}_M^i(\tau) \cup \bigcup_{(g_1,\ldots,g_k)} \left\{ \begin{array}{l} (\tau, (g_1, o_1), \ldots, (g_k, o_k), n) : \\ \quad o_1 \in \mathcal{U}_M^i(a_1), \ldots, o_k \in \mathcal{U}_M^i(a_k), n \in \{1, 2, 3, \ldots\} \end{array} \right\}
\end{aligned}$$

where $(g_1, \ldots, g_k)$ ranges over the origin function tuples for number statements of type τ in M, and $a_1, \ldots, a_k$ are the return types of $g_1, \ldots, g_k$. The universe *for type τ is:*

$$\mathcal{U}_M(\tau) \triangleq \bigcup_{i=0}^{\infty} \mathcal{U}_M^i(\tau)$$

Note that if $\mathrm{Guar}_M(\tau)$ is countable — which is true for all types in discrete BLOG — then $\mathcal{U}_M(\tau)$ is countable as well. We can now define the set of possible worlds for a BLOG model.

Definition 4.10. *For a BLOG model M, the set of possible worlds Ω_M is the set of model structures ω of $\mathcal{L}_M$ such that:*

i. for each type τ, $\mathrm{Guar}_M(\tau) \subseteq [\tau]^\omega \subseteq \mathcal{U}_M(\tau)$;

ii. for each nonrandom function f in M with type signature $(r, a_1, \ldots, a_k)$, and each tuple of argument values $(o_1, \ldots, o_k) \in [a_1]^\omega \times \cdots \times [a_k]^\omega$,

$$[f]^\omega(o_1, \ldots, o_k) = \begin{cases} [f]_M(o_1, \ldots, o_k) & \text{if } o_i \in \mathrm{Guar}_M(a_i) \text{ for } i \in \{1, \ldots, k\} \\ \mathsf{null} & \text{otherwise;} \end{cases}$$

iii. for each origin function g in M with argument type τ, and each argument value $q \in [\tau]^\omega$,

$$[g]^\omega(q) = \begin{cases} o & \text{if } q \text{ has the form } (\tau, \ldots, (g, o), \ldots) \text{ for some } o \in [\mathrm{ret}(g)]^\omega \\ \mathsf{null} & \text{otherwise;} \end{cases}$$

iv. for each number statement in M with type τ and origin functions $g_1, \ldots, g_k$, and each tuple of generating objects $(o_1, \ldots, o_k) \in [\mathrm{ret}(g_1)]^\omega \times \cdots \times [\mathrm{ret}(g_k)]^\omega$, there is a natural number N (possibly zero) such that:

$$\{n \in \mathbb{N} : (\tau, (g_1, o_1), \ldots, (g_k, o_k), n) \in [\tau]^\omega\} = \{1, \ldots, N\}.$$

Part (i) of this definition ensures that the guaranteed objects exist in each possible world. It also asserts that for each type τ, there are no other objects besides those in $\mathcal{U}_M(\tau)$. This is a kind of *domain closure* assumption, but a very weak one: unlike standard domain closure assumptions [Reiter, 1980], it does not require that the objects be referred to by constant symbols or even ground terms in the language. Part (ii) of this definition ensures that the nonrandom functions have the desired interpretations. On argument tuples that consist solely of guaranteed objects, a nonrandom function yields the value specified by its definition in the model; on other tuples, it yields null.

Part (iii) specifies the interpretations of origin functions g on objects q. If q is a tuple $(\tau, \ldots, (g, o), \ldots)$, then $[g]^\omega(q) = o$ in every world ω where q exists. Otherwise, $[g]^\omega(q) = \mathsf{null}$. Thus origin functions always yield null on guaranteed objects, which are not tuples of the form $(\tau, \ldots)$. On non-guaranteed objects, the definition is unambiguous because a number statement cannot use the same origin function more than once (and every non-guaranteed object in $\mathcal{U}_M(\tau)$ corresponds to an application of some number statement). Another consequence of this assertion is that if $(\tau, (g_1, o_1), \ldots, (g_k, o_k), n) \in [\tau]^\omega$, then its generating objects $o_1, \ldots, o_k$ must exist in ω as well. This follows from the definition of a model structure (Definition 2.6): the values of functions (in this case, origin functions) in ω must be objects that exist in ω. Finally, part (iv) implies that in each world, the objects satisfying a given application of a number statement are numbered $1, 2, \ldots, N$ for some finite N. This

rules out having an infinite set of objects that satisfy a given number statement application. It also implies that if we know how many objects satisfy a number statement application, then we know which objects they are.

Note that although we refer to Ω_M as the set of "possible worlds" of model M, the model may assign probability zero to some of these worlds. For instance, if M is the urn-and-balls model shown in Figure 4.1, then Ω_M includes worlds ω where $[\mathsf{Ball}]^\omega$ is non-empty but $[\mathsf{BallDrawn}]^\omega(\mathsf{Draw1}) = \mathsf{null}$, even though such worlds have probability zero under the model. This choice of definitions means that changing the dependency statements in a BLOG model cannot change the set of possible worlds; it only changes the probability distribution over these worlds.

Now that we have defined the outcome space for a BLOG model, we will give a formal definition of the set of basic random variables.

Definition 4.11. *For a BLOG model M, the set $\mathcal{V}_M$ of* basic random variables *consists of:*

- *for each random function f in M with type signature $(r, a_1, \ldots, a_k)$ and each tuple of arguments $(o_1, \ldots, o_k) \in \mathcal{U}_M(a_1) \times \cdots \times \mathcal{U}_M(a_k)$, a* function application variable*:*

$$V_f[o_1, \ldots, o_k](\omega) \triangleq \begin{cases} [f]^\omega(o_1, \ldots, o_k), \text{ if } o_i \in [a_i]^\omega \text{ for } i \in \{1, \ldots, k\} \\ \mathsf{null}, \text{ otherwise} \end{cases}$$

- *for each number statement with type τ and origin functions $g_1, \ldots, g_k$ that have return types $\tau_1, \ldots, \tau_k$, and each tuple of objects $(o_1, \ldots, o_k) \in \mathcal{U}_M(\tau_1) \times \cdots \times \mathcal{U}_M(\tau_k)$, a* number variable $N_\tau[g_1 = o_1, \ldots, g_k = o_k](\omega)$ *equal to the number of objects that satisfy this number statement applied to $o_1, \ldots, o_k$ in ω.*

For functions with no arguments and number statements with no origin functions, we will omit the empty brackets when referring to the corresponding basic variables.

Intuitively, each step in the generative world-construction process determines the value of a basic variable. The range of a function application variable for a function with return type r is $\mathcal{O}_M(r) \cup \{\mathsf{null}\}$, and the range of a number variable is $\mathbb{N}$. Note that if the objects $o_1, \ldots, o_k$ do not exist in a world ω, then any number variable $N_\tau\,[g_1 = o_1, \ldots, g_k = o_k]$ has the value zero in ω, because no objects can satisfy that number variable in ω.

4.3.3 Achievable instantiations of basic random variables

It is fairly obvious that in any BLOG model, a complete instantiation of the basic random variables corresponds to at most one possible world. Such an instantiation specifies the interpretations of the random functions and the number of objects satisfying each number statement application; given these constraints, there is at most one model structure that satisfies Definition 4.10. However, some instantiations are not *achievable*: they do not correspond to any model structure. For instance, in the urn-and-balls model, the instantiation ($N_{\mathsf{Ball}} = 4$, $V_{\mathsf{TrueColor}}\,[(\mathsf{Ball}, 8)] = \mathsf{Blue}$) is unachievable, because $(\mathsf{Ball}, 8)$ cannot exist in a world ω where $N_{\mathsf{Ball}} = 4$, and $V_{\mathsf{TrueColor}}\,[(\mathsf{Ball}, 8)]$ must yield null on worlds where $(\mathsf{Ball}, 8)$ does not exist. Another unachievable instantiation is ($N_{\mathsf{Ball}} = 4$, $V_{\mathsf{BallDrawn}}\,[\mathsf{Draw1}] = (\mathsf{Ball}, 8)$), in which the value of $V_{\mathsf{BallDrawn}}\,[\mathsf{Draw1}]$ is an object that does not exist. A less obvious example is the infinite instantiation ($V_{\mathsf{TrueColor}}\,[(\mathsf{Ball}, 1)] = \mathsf{Blue}$, $V_{\mathsf{TrueColor}}\,[(\mathsf{Ball}, 2)] = \mathsf{Blue}$, $V_{\mathsf{TrueColor}}\,[(\mathsf{Ball}, 3)] = \mathsf{Blue}, \ldots$). This instantiation is unachievable because there is no possible world in which infinitely many balls exist. Any completion of this instantiation would have to assert $N_{\mathsf{Ball}} = n$ for some finite number n, contradicting the assertions $V_{\mathsf{TrueColor}}\,[(\mathsf{Ball}, m)] = \mathsf{Blue}$ for all $m > n$.

Fortunately, there is a reasonably simple way to check that an instantiation (complete or partial) on the $\mathcal{V}_M$ is achievable. To state this condition, we need to introduce

some terminology. If o is a non-guaranteed object $(\tau, (g_1, o_1), \ldots, (g_k, o_k), n)$, then its *governing number variable*, denoted N_o, is $N_\tau[g_1 = o_1, \ldots, g_k = o_k]$. The number n in the tuple representation of o will be denoted $\text{index}(o)$.

An instantiation σ *uses* an object o *as an argument* if $\text{vars}(\sigma)$ includes a number variable N with o as an argument such that $\sigma_N \neq 0$, or a function application variable V with o as an argument such that $\sigma_V \neq \mathsf{null}$. The instantiation σ *uses* o *as a value* if there is some $X \in \text{vars}(\sigma)$ such that $\sigma_X = o$. We will also say simply that an instantiation *uses* o if it uses o as an argument or value.

Definition 4.12. *In a BLOG model M, an instantiation σ on $\mathcal{V}_M$ is* governing-variable complete *if, for every non-guaranteed object o that σ uses, σ instantiates N_o to a value greater than or equal to* $\text{index}(o)$.

Lemma 4.1. *In any BLOG model M, if an instantiation σ on $\mathcal{V}_M$ is governing-variable complete, then it is achievable.*

Proof. Suppose σ is governing-variable complete. We can construct a model structure ω consistent with σ as follows. For each type τ, let $[\tau]^\omega$ consist of:

1. the guaranteed objects $\text{Guar}_M(\tau)$; and
2. for each number variable $N = N_\tau[g_1 = o_1, \ldots, g_k = o_k]$ in $\text{vars}(\sigma)$, the set of non-guaranteed objects $\{(\tau, (g_1, o_1), \ldots, (g_k, o_k), n) \ : \ n \in \{1, \ldots, \sigma_N\}\}$ (note that this is an empty set if $\sigma_N = 0$).

Thus, uninstantiated number variables get no satisfiers. Let the interpretations of the nonrandom functions and origin functions in ω be as specified in Definition 4.10. Finally, for each random function symbol f with argument types $(a_1, \ldots, a_k)$, let $[f]^\omega$ be the function on tuples $(o_1, \ldots, o_k) \in [a_1]^\omega \times \cdots \times [a_k]^\omega$ such that:

$$[f]^\omega(o_1, \ldots, o_k) = \begin{cases} \sigma_{V_f[o_1,\ldots,o_k]} & \text{if } V_f[o_1, \ldots, o_k] \in \text{vars}(\sigma) \\ \mathsf{null} & \text{otherwise} \end{cases}$$

So function application variables that are not instantiated are treated as if they were set to null.

First we will check that this ω is indeed a model structure: that is, for each function symbol f, the values of $[f]^\omega$ are all in $[\mathrm{ret}(f)]^\omega \cup \{\mathsf{null}\}$. For nonrandom functions f, this is ensured by the definition of $[f]_M$. Now consider any origin function g with argument type τ, and any object $o \in [\tau]^\omega$ such that $[g]^\omega(o) \neq \mathsf{null}$. Since the interpretations of origin functions in ω are given by Definition 4.10(iii), we know that o is of the form $(\tau, (g_1, o_1), \ldots, (g_k, o_k), n)$, and $[g]^\omega(o) = o_i$ for some $i \in \{1, \ldots, k\}$. Given our construction of ω, the fact that o is in $[\tau]^\omega$ implies that σ instantiates N_o to a value greater than or equal to n. But N_o has o_i as an argument, so this means σ uses o_i as an argument. Therefore, since σ is governing-variable complete, it also instantiates $N_{(o_i)}$ to a value greater than or equal to $\mathrm{index}(o_i)$. So $o_i \in [r]^\omega$.

Now consider any random function f and any argument tuple $(o_1, \ldots, o_k)$ such that $[f]^\omega(o_1, \ldots, o_k) = q \neq \mathsf{null}$. Then our construction ensures that $q = \sigma_{V_f[o_1,\ldots,o_k]}$. So σ uses q as a value. If q is a guaranteed object then it is automatically in $[\mathrm{ret}(f)]^\omega$; otherwise, the fact that σ is governing-variable complete implies that σ must instantiate N_q to a value greater than or equal to $\mathrm{index}(q)$. So by construction, $q \in [r]^\omega$.

It is easy to check that ω is in Ω_M, as defined in Definition 4.10. Our construction also makes it clear that for each $X \in \mathrm{vars}(\sigma)$, $X(\omega) = \sigma_X$. So ω is a world in Ω_M that is consistent with σ. □

An instantiation on $\mathcal{V}_M$ may be achievable even if it is not governing-variable complete: for example, the instantiation $(V_{\mathsf{TrueColor}}[(\mathsf{Ball}, 1)] = \mathsf{Blue})$ is achievable. Thus, the converse of this lemma does not hold. However, we can use it to show that in every BLOG model M, the basic variables are sufficient for $(\Omega_M, \mathscr{F}_M)$, in

the sense of Definition 3.2.

Lemma 4.2. *In any discrete BLOG model M, the set of basic random variables $\mathcal{V}_M$ is sufficient for $(\Omega_M, \mathscr{F}_M)$.*

Proof. First we must show that $\mathscr{F}_M$ is generated by events of the form $\{X = x\}$ for $X \in \mathcal{V}_M$. It suffices to show that these events generate the basis events for $\mathscr{F}_M$ described in Definition 4.7. First, for any type τ in $\mathcal{L}_M$ and any object $o \in \mathcal{O}_M(\tau)$, consider the event $\{\omega \in \Omega_M \ : \ o \in [\tau]^\omega\}$. If $o \in \mathrm{Guar}_M(\tau)$, then this event is simply Ω_M. Otherwise, Definitions 4.10(i) and 4.9 imply that o has the form $(\tau, (g_1, o_1), \ldots, (g_k, o_k), n)$. Then by Definition 4.10(iv), $\{\omega \in \Omega_M \ : \ o \in [\tau]^\omega\} = \{N_\tau [g_1 = o_1, \ldots, g_k = o_k] \geq n\}$. This event is clearly generated by events of the form $\{N_\tau [g_1 = o_1, \ldots, g_k = o_k] = x\}$. Now consider the other basis events for $\mathscr{F}_M$, which have the form $\{\omega \in \Omega_M \ : \ o_1 \in [a_1]^\omega, \ldots, o_k \in [a_k]^\omega \text{ and } [f]^\omega (o_1, \ldots, o_k) = o\}$. In the case where $o \neq \mathsf{null}$, each such event is equal to $\{V_f [o_1, \ldots, o_k] = o\}$. For $o = \mathsf{null}$, we need to be more careful, because null serves as a value for $V_f [o_1, \ldots, o_k]$ when the function takes the value null and when one of the arguments does not exist. However, we have already shown how to generate events of the form $\{\omega \in \Omega_M \ : \ o_i \in [a_i]^\omega\}$. So we can generate the desired events as follows:

$$\begin{aligned}\{\omega \in \Omega_M \ : \ o_1 \in [a_1]^\omega, \ldots, o_k \in [a_k]^\omega \text{ and } [f]^\omega (o_1, \ldots, o_k) = o\} \\ = \{V_f [o_1, \ldots, o_k] = \mathsf{null}\} \cap \bigcap_{i=1}^{k} \{\omega \in \Omega_M \ : \ o_i \in [a_i]^\omega\}\end{aligned}$$

For the second part of the definition of a sufficient set of variables, consider any complete instantiation σ of $\mathcal{V}_M$ of which each finite sub-instantiation is acheivable. We will show that σ itself is achievable. By Lemma 4.1, it suffices to show that σ is governing-variable complete. So consider any non-guaranteed object o that σ uses. Because σ is complete, it instantiates N_o to some value. Assume for contradiction

that $\sigma_{N_o} < \text{index}(o)$. Then let τ be the restriction of σ to the variables $\{N_o, X\}$, where X is any variable in $\text{vars}(\sigma)$ that uses o. Since τ is a finite sub-instantiation of σ, it must be achievable; let ω be any world consistent with τ. Because $N_o(\omega) < \text{index}(o)$, N_o must have fewer than $\text{index}(o)$ satisfiers in ω. Given this fact, parts (iii) and (iv) of Definition 4.10 imply that o does not exist in ω.

Now suppose the variable X that uses o is a number variable. This means that X has o as an argument and $\sigma_X > 0$. By Definitions 4.11 and 4.8, this implies that ω contains a nonzero number of objects that are mapped to o by some origin function g. But this is impossible, because $[g]^\omega$ can only map arguments to values that exist in ω. We get a similar result if X is a function application variable $V_f[o_1, \ldots, o_k]$ that uses o. Then either $\sigma_X = o$, or $o_i = o$ for some $i \in \{1, \ldots, k\}$ and $\sigma_X \neq \mathsf{null}$. The first option implies $[f]^\omega(o_1, \ldots, o_k) = o$, which violates the definition of a model structure since o does not exist in ω. The second option violates 4.11, which asserts that $V_f[o_1, \ldots, o_k](\omega) = \mathsf{null}$ when one of $o_1, \ldots, o_k$ does not exist in ω. So in all cases, we have a contradiction. □

Thus, every BLOG model M defines a set of basic random variables $\mathcal{V}_M$ that is sufficient for $(\Omega_M, \mathscr{F}_M)$. We now move on to discuss the probability model that M defines over these variables.

4.3.4 The PBM defined by a BLOG model

The dependency statements and number statements of a BLOG model define conditional probability distributions (CPDs) for the basic random variables. Specifically, the dependency statement for a random function f defines CPDs for all function application variables that involve f, and the number statement with origin functions $(g_1, \ldots, g_k)$ defines CPDs for all number variables involving that tuple of origin functions. But the dependency statement for a variable X is made up of formulas and

other BLOG expressions; it does not directly say which basic variables X depends on, or under what conditions these dependencies are active.

Of course, we could define rules for extracting such variable-level dependencies from formulas, terms, and set expressions. However, the partition-based models (PBMs) defined in Section 3.4 allow us to define the semantics of dependency statements at a more fundamental level. For each random variable X, a PBM specifies a set of events that form a partition of the outcome space, as well as a conditional probability for each value of X given each event in that partition. In a BLOG model M, the dependency statement for a basic variable X (with the logical variables in its header set equal to X's arguments) defines such a partition Λ_M^X and CPD c_M^X as follows. Each partition block is a set of worlds where a particular clause i is the first clause whose condition is satisfied, and the CPD arguments in clause i have some particular values $q_{i1}, \ldots, q_{in_i}$. The probability of X taking a value x given this event is $c_i(x, (q_{i1}, \ldots, q_{in_i}))$, where c_i is the elementary CPD in clause i.

To formalize this definition, recall from Section 4.2.5 that each dependency statement or number statement defines a scope β and a sequence of clauses of the form $\{(\varphi_i, c_i, (e_{i1}, \ldots, e_{i(n_i)}))\}_{i=1}^m$. The scope β consists of the variables introduced to represent function arguments in a dependency statement or generating objects in a number statement; the formulas φ_i and arguments e_{ij} are well-formed in this scope. Also, the condition φ_m in the last clause is always simply true. Given a possible world ω and an assignment α whose domain is β, the *active clause index* is the first index i such that $\omega \models_\alpha \varphi_i$. Note that some such index always exists because $\varphi_m = \mathsf{true}$.

Definition 4.13. *In a BLOG model M, let X be a function application variable $V_f[o_1, \ldots, o_k]$ or a number variable $N_\tau[g_1 = o_1, \ldots, g_k = o_k]$. Consider the dependency statement for f or the number statement for τ with origin functions $(g_1, \ldots, g_k)$.*

Let $\{(x_1, \tau_1), \ldots, (x_k, \tau_k)\}$ *be the scope defined by the header of this number statement, and* α *be the assignment that maps* (x_i, τ_i) *to* o_i *for* $i \in \{1, \ldots, k\}$*. Suppose this statement defines clauses* $\{(\varphi_i, c_i, (e_{i1}, \ldots, e_{i(n_i)}))\}_{i=1}^{m}$*. Then let* $\sim$ *be the equivalence relation on* Ω_M *such that* $\omega_1 \sim \omega_2$ *if:*

- α *is invalid in both* ω_1 *and* ω_2*, or;*
- α *is valid in both* ω_1 *and* ω_2*,* ω_1 *and* ω_2 *have the same active clause index* i *given* α*, and* $[e_{ij}]_\alpha^{\omega_1} = [e_{ij}]_\alpha^{\omega_2}$ *for* $j \in \{1, \ldots, n_i\}$*.*

Then Λ_M^X *is the set of equivalence classes* $\Omega_M/\sim$*, and the CPD* c_M^X *is defined as follows:*

- *for the block* $\lambda_0 \in \Lambda_M^X$ *(if any) such that* α *is invalid in all worlds in* λ_0*,*

$$c_M^X(x, \lambda_0) = \begin{cases} \delta(x, \mathsf{null}) & \text{if } X \text{ is a function application variable} \\ \delta(x, 0) & \text{if } X \text{ is a number variable} \end{cases}$$

- *for each other block* λ *where for all* $\omega \in \lambda$*, the active clause index is* i *and* $[e_{ij}]_\alpha^{\omega} = q_j$ *for* $j \in \{1, \ldots, n_i\}$*,*

$$c_M^X(x, \lambda) = c_i\left(x, (q_1, \ldots, q_{n_i})\right).$$

To see how this definition works, let's start with a very simple dependency statement:

```
TrueColor(b) ~ TabularCPD[[0.5, 0.5]];
```

Consider a particular ball $(\mathsf{Ball}, 3)$. To get the partition and CPD for the function application variable $X = V_{\mathsf{TrueColor}}[(\mathsf{Ball}, 3)]$, we use the assignment $\alpha = ((b, \mathsf{Ball}) \mapsto$

$(\mathsf{Ball}, 3)$). Now, this dependency statement has only one clause and no CPD arguments. So Λ_M^X has just two blocks: one where α is valid, and one where it is not. The block where α is invalid corresponds to the event that $(\mathsf{Ball}, 3)$ does not exist. This is the same as the event $\{N_{\mathsf{Ball}}(\omega) < 3\}$, but we did not have to define this variable-level dependency explicitly; it falls out of our general definition. Given that α is invalid, the CPD c_M^X gives probability one to null, which is the value that function application variables take on when their arguments do not exist. Given the other block, where α is valid, c_M^X puts probability 0.5 on each of the two guaranteed Color objects, Blue and Green. Note that the partition defined by this dependency statement depends on the value assigned to b. Thus, a dependency statement does not just define a single partition: it is a first-order specification that maps logical variable assignments to partitions and CPDs.

As a slightly more complicated example, consider the dependency statement:

```
ObsColor(d)
    if (BallDrawn(d) != null) then
        ~ TabularCPD[[0.8, 0.2], [0.2, 0.8]]
            (TrueColor(BallDrawn(d)));
```

For the basic variable $X = V_{\mathsf{ObsColor}}[(\mathsf{Draw}, 1)]$, we use the assignment $\alpha = ((d, \mathsf{Draw}) \mapsto (\mathsf{Draw}, 1))$. This assignment is valid in all possible worlds, since $(\mathsf{Draw}, 1)$ is a guaranteed object. So there is no partition block where α is invalid. There are three partition blocks where the first clause is active: these correspond to the events $\{\omega \models_\alpha (\mathsf{BallDrawn}(d) \neq \mathsf{null})\} \cap \{[\mathsf{TrueColor}(\mathsf{BallDrawn}(d))]_\alpha^\omega = c\}$, for $c \in \{\mathsf{Blue}, \mathsf{Green}, \mathsf{null}\}$. Finally, there is a partition block where the implicit `else` clause is active: this is the event $\{\omega \models_\alpha (\mathsf{BallDrawn}(d) = \mathsf{null})\}$.

As a whole, the urn-and-balls BLOG model in Figure 4.1 defines the same PBM that we described explicitly in Figure 3.6. In fact, the BLOG model does more than

Figure 3.6 does: not only does it specify partitions and CPDs, but it also specifies the outcome space (via Definition 4.10) and defines the random variables on this outcome space (via Definition 4.11). We used two paragraphs of text and equations to define these aspects of the model in Section 3.1. Even as a means for describing the model to another human being, the BLOG model is much clearer and more concise than our earlier presentation; on top of that, the BLOG model has a formal, machine-readable syntax.

The following proposition affirms that the partitions and CPDs defined in Definition 4.13 indeed satisfy the definition of a PBM.

Proposition 4.3. *For any discrete BLOG model M, there is a PBM Γ_M over $\mathcal{V}_M$ such that for each variable $X \in \mathcal{V}_M$, $\Lambda^X_{\Gamma_M} = \Lambda^X_M$ and $c^X_{\Gamma_M} = c^X_M$.*

Proof. The definition of a PBM (Definition 3.5) requires that $\mathcal{V}_M$ be sufficient for the outcome space on which they are defined, that each partition Λ^X_M be measurable and countable, and that each function c^X_M satisfy the definition of a CPD. The fact that $\mathcal{V}_M$ is sufficient for $(\Omega_M, \mathscr{F}_M)$ is given by Lemma 4.2. The fact that the blocks in each partition Λ^X_M are measurable follows directly from Lemma 4.17, which asserts that the event in which a BLOG expression takes on a particular value is always measurable. Furthermore, any expression in discrete BLOG denotes either an object in the countable universe $\mathcal{U}_M(\tau)$ for some type τ, or a finite set of such objects, or a finite multiset of tuples of such objects, or null. Thus, the set of all possible values of expressions in a given discrete BLOG model is countable. A block in Λ^X_M (other than the "assignment invalid" block) is defined by an index i and a finite tuple $(q_1, \ldots, q_{n_i})$ of such values, so Λ^X_M is countable as well. Finally, the definition of an elementary CPD ensures that for any $X \in \mathcal{V}_M$ and $\lambda \in \Lambda^X_M$, the function $c^X_M(x, \lambda)$ yields values in $[0, 1]$ and sums to one over $x \in \text{range}(X)$. Thus, c^X_M is indeed a CPD for X given Λ^X_Γ, as defined by Definition 3.4. □

We can now define what it means for a probability measure to satisfy a BLOG model.

Definition 4.14. *A probability measure P on $(\Omega_M, \mathscr{F}_M)$* satisfies *a discrete BLOG model M if it satisfies Γ_M. A BLOG model M is* well-defined *if there is exactly one probability measure that satisfies it.*

4.3.5 Ruling out unachievable instantiations

Theorem 3.13 states that a PBM is well-defined if it satisfies two conditions: it respects its outcome space, and each of its outcomes has a supportive numbering. Intuitively, a PBM respects its outcome space (Definition 3.12) if any instantiation of the basic variables that has a supportive numbering but is not achievable has probability zero. The other requirement, that every outcome have a supportive numbering, is a less restrictive, context-specific version of the requirement that a Bayesian network have a topological numbering.

It is possible to write down BLOG models whose possible worlds do not have supportive numberings. However, every discrete BLOG model respects its outcome space. To see why this is true, recall that by Lemma 4.1, any instantiation on the basic random variables that is governing-variable complete is achievable. Thus, the only way an instantiation can fail to be achievable is if it uses a non-guaranteed object o without instantiating the governing number variable N_o to a value greater than or equal to $\text{index}(o)$. We will show that if an instantiation with a supportive numbering does this, then it must have probability zero. We begin with the following lemmas:

Lemma 4.4. *In a discrete BLOG model M, let X be a basic random variable with function arguments or generating objects $o_1, \ldots, o_k$ of types $\tau_1, \ldots, \tau_k$. Suppose $\lambda \in \Lambda_M^X$ and $o \in \text{range}(X)$. If $c_M^X(o, \lambda) > 0$, then:*

i. if X is a function application variable and $o \neq \mathsf{null}$, or X is a number variable and $o \neq 0$, then $o_i \in [\tau_i]^\omega$ for all $i \in \{1, \ldots, k\}$ and all $\omega \in \lambda$;

ii. if o is a non-guaranteed object of type τ, then $o \in [\tau]^\omega$ for all $\omega \in \lambda$.

Proof. For part (i), let $x_1, \ldots, x_k$ be the logical variables introduced in the header of the the dependency or number statement for X. Let α be the assignment that maps (x_i, τ_i) to o_i for $i \in \{1, \ldots, k\}$. Recall from Definition 4.13 that Λ_M^X includes at most one block λ_0 containing worlds where α is invalid — that is, where some of the objects $o_1, \ldots, o_k$ do not exist. Given λ_0, c_M^X assigns probability one to the value null for function application variables, and the value 0 for number variables. So if $c_M^X(o, \lambda) > 0$ and o is not this "invalidity" value, then $\lambda \neq \lambda_0$. Therefore α is valid in all worlds in λ.

For part (ii), suppose o is a non-guaranteed object of type τ. Then the fact that $c_M^X(o, \lambda) > 0$ implies $\lambda \neq \lambda_0$. So λ corresponds to the event that α is valid, the active clause index in the dependency or number statement is some index i, and the CPD arguments e_{ij} have some particular denotations q_j for $j \in \{1, \ldots, n_i\}$. The definition of an elementary CPD stipulates that if o is a non-guaranteed object and $c_i(o, (q_1, \ldots, q_{n_i})) > 0$, then one of the argument values $q_1, \ldots, q_{n_i}$ contains o. It is easy to see from the definitions of the denotations of terms (Definition 2.7) and set expressions (Definition 4.3) that if an assignment α is valid in a world ω, then the denotation $[e]_\alpha^\omega$ of any expression e contains only objects that exist in ω. Thus $q_1, \ldots, q_{n_i}$ contain only objects that exist in all worlds $\omega \in \lambda$. So since $c_M^X(o, \lambda) > 0$, o must exist in all these worlds. □

Lemma 4.5. *Let M be a BLOG model and σ be an instantiation on $\mathcal{V}_M$ that is governing-variable complete. Let o be a non-guaranteed object of type τ. If $o \in [\tau]^\omega$ for all $\omega \in \mathrm{ev}(\sigma)$, then σ instantiates N_o to a value greater than or equal to* $\mathrm{index}(o)$.

Proof. We will prove the contrapositive: if σ does not instantiate N_o to a value greater than or equal to index (o), then there is some $\omega \in \text{ev}(\sigma)$ such that $o \notin [\tau]^\omega$. First suppose σ instantiates N_o to a value less than index (o). Then the definition of a number variable and parts (iii) and (iv) of Definition 4.10 imply that $o \notin [\tau]^\omega$ for each $\omega \in \text{ev}(\sigma)$. And since σ is governing-variable complete, Lemma 4.1 guarantees that it is achievable, so at least one such ω exists. For the remaining case, suppose σ does not instantiate N_o at all. Then let $\sigma' = (\sigma; N_o = 0)$. Because a number variable that is instantiated to zero does not use its arguments (by the definition of "use" that we gave before Definition 4.12), σ' uses the same set of objects as σ. Thus σ' is also governing-variable complete. So, as we argued for the first case, there is a world $\omega \in \text{ev}(\sigma')$ such that $o \notin [\tau]^\omega$. Since $\text{ev}(\sigma') \subseteq \text{ev}(\sigma)$, this ω is also in $\text{ev}(\sigma)$. □

Proposition 4.6. *If M is a discrete BLOG model, then Γ_M respects its outcome space.*

Proof. By the definition of respecting an outcome space, we must show that if π is a supportive numbering of an unachievable instantiation σ, then π yields a zero factor on σ: that is, one of the factors $c_M^X(\sigma_X, \lambda_\Gamma^X(\sigma[\text{Pred}_\pi[X]]))$ is equal to zero. We will show the contrapositive: that if π does not yield a zero factor on σ, then σ is achievable. We begin with an inductive proof that if σ is a finite instantiation with a supportive numbering that does not yield a zero factor, then σ is governing-variable complete, and thus achievable by Lemma 4.1.

The base case, where $\sigma = \top$, is trivial because $\top$ does not use any objects. Now suppose the claim holds for instantiations of size n, and let π be a supportive numbering that does not yield a zero factor on an instantiation σ of size $n+1$. Let Y be the last variable in vars (σ) according to π. Since π is a supportive numbering, we know σ_{-Y} supports Y. Also, since π does not yield a zero factor on σ, we know $c_M^Y(\sigma_Y, \lambda_M^Y(\sigma_{-Y})) > 0$.

Now consider any non-guaranteed object o that is used by σ but not by σ_{-Y}. Such an object must be either the value σ_Y, or an argument of the basic variable Y. If σ_Y is a non-guaranteed object o of type τ, then by Lemma 4.4(ii), $o \in [\tau]^\omega$ for all $\omega \in \lambda_M^Y(\sigma_{-Y})$. Similarly, if the instantiation $(Y = \sigma_Y)$ uses o as an argument, then Lemma 4.4(i) implies $o \in [\tau]^\omega$ for all $\omega \in \lambda_M^Y(\sigma_{-Y})$. In either case, since $\mathrm{ev}\,(\sigma_{-Y}) \subseteq \lambda_M^Y(\sigma_{-Y})$, we have $\sigma_Y \in [\tau]^\omega$ for all $\omega \in \mathrm{ev}\,(\sigma_{-Y})$. We also know by the inductive hypothesis that σ_{-Y} is governing-variable complete. So by Lemma 4.5, σ_{-Y} must instantiate N_o to a value greater than or equal to $\mathrm{index}\,(o)$. Thus, σ is governing-variable complete as well.

This inductive argument applies to all finite instantiations. Now suppose π is a supportive numbering that does not yield a zero factor on an infinite instantiation σ. Consider any object o that is used by σ. This object must be used by some variable; let X be the first variable (according to π) that uses it. Then the instantiation σ' obtained by restricting σ to X is a finite instantiation with a supportive numbering; hence, it is governing-variable complete. Since σ' uses o, it must instantiate N_o to a value greater than or equal to $\mathrm{index}\,(o)$. Therefore σ instantiates N_o to the same value. Since this holds for all objects that σ uses, σ is governing-variable complete. So by Lemma 4.1, it is achievable. □

4.3.6 Well-defined BLOG models

Given the results of the previous section, it is now straightforward to prove our main theorem about BLOG. This theorem uses the notions of a supportive numbering and a self-supporting instantiation from Section 3.4. An instantiation σ *supports* a variable X if it picks out a unique block in Λ_M^X (Definition 3.6). In other words, σ supports X if all the worlds in $\mathrm{ev}\,(\sigma)$ agree on the active clause index in X's dependency/number statement and the denotations of the CPD arguments in this clause

(or if each world in σ denies the existence of some arguemnt of X). Thus, for example, the instantiation $(V_{\mathsf{BallDrawn}}[\mathsf{Draw1}] = (\mathsf{Ball}, 3), V_{\mathsf{TrueColor}}[(\mathsf{Ball}, 3)] = \mathsf{Blue})$ supports the variable $V_{\mathsf{ObsColor}}[\mathsf{Draw1}]$. An instantiation is *self-supporting* (Definition 3.7) if it supports every variable it instantiates; self-supporting instantiations are analogous to instantiations of ancestral sets in a Bayesian network. A supportive numbering of a world ω is a numbering of $\mathcal{V}_M$ such that for each $X \in \mathcal{V}_M$, the instantiation of $\mathrm{Pred}_\pi[X]$ in ω supports X. Supportive numberings are analogous to topological numberings of the variables in a BN, but they can be specific to particular worlds.

Theorem 4.7. *Let M be a discrete BLOG model in which each possible world $\omega \in \Omega_M$ has a supportive numbering. Then M is well-defined: that is, there is a unique probability measure P_M on $(\Omega_M, \mathscr{F}_M)$ such that for each basic variable $X \in \mathcal{V}_M$,*

- *c_M^X is a version of $P_M(X|\Lambda_M^X)$;*
- *for every finite, self-supporting instantiation σ on $\mathcal{V}_M$ that does not instantiate X, $X \perp\!\!\!\perp_{(P_M)} \sigma \,|\, \Lambda_M^X$.*

Proof. By Proposition 4.3, M defines a PBM Γ_M. By Proposition 4.6, this PBM respects its outcome space. So if every possible world has a supportive numbering, then Theorem 3.13 ensures that Γ_M is well-defined. This is the same as saying that M is well-defined. The remainder of the theorem simply reiterates the definition of a probability measure that satisfies a PBM from Definition 3.8. □

The probability measure P_M described in this theorem is the probability measure represented by the BLOG model M. The theorem reiterates what it means for a probability measure to satisfy a PBM in order to emphasize the declarative nature of BLOG models: like a BN, a BLOG model specifies a CPD for each variable, along with certain conditional independence properties.

Theorem 4.7 requires that every possible world ω have a supportive numbering. All the BLOG models we have used as examples satisfy this condition. In the urn-and-balls model of Figure 4.1, we can construct a supportive numbering for any possible world ω as follows. First take N_{Ball}, then the variables $V_{\mathsf{TrueColor}}[(\mathsf{Ball}, i)]$ for $i \in \{1, \ldots, N_{\mathsf{Ball}}(\omega)\}$. Then take $V_{\mathsf{BallDrawn}}[d]$ followed by $V_{\mathsf{ObsColor}}[d]$ for each draw d. Finally, take the remaining basic variables — the variables $V_{\mathsf{TrueColor}}[(\mathsf{Ball}, i)]$ for $i > N_{\mathsf{Ball}}(\omega)$ — in any order. Note that there is no single numbering that is supportive for all worlds: such a numbering would have to put all the infinitely many TrueColor variables before the first ObsColor variable, because for every ball (Ball, i), there is some world where BallDrawn(Draw1) denotes (Ball, i).

In the hurricane model of Figure 4.2, a supportive numbering for a world ω begins with V_{First}. If $V_{\mathsf{First}}(\omega) = \mathsf{A}$, then $V_{\mathsf{Prep}}[\mathsf{A}]$ and $V_{\mathsf{Damage}}[\mathsf{A}]$ come next, followed by $V_{\mathsf{Prep}}[\mathsf{B}]$ and then $V_{\mathsf{Damage}}[\mathsf{B}]$. If $V_{\mathsf{First}}(\omega) = \mathsf{B}$, then the variables defined on B precede the variables defined on A.

Now consider the citation model of Figure 4.3. A supportive numbering for a world ω starts with $N_{\mathsf{Researcher}}$, followed by $V_{\mathsf{Name}}[(\mathsf{Researcher}, i)]$ for $i \leq N_{\mathsf{Researcher}}(\omega)\}$. Next comes $N_{\mathsf{Publication}}$. Then, we can take the variables $V_{\mathsf{Title}}[(\mathsf{Publication}, i)]$ and $V_{\mathsf{NumAuthors}}[(\mathsf{Publication}, i)]$ for $i < N_{(\mathsf{Publication})}(\omega)$. The next variables to take are $V_{\mathsf{NthAuthor}}[(\mathsf{Publication}, i), n]$ for pairs i, n such that $i \leq N_{(\mathsf{Publication})}(\omega)$ and $n < V_{\mathsf{NumAuthors}}[(\mathsf{Publication}, i)](\omega)$. Then for each citation c, take $V_{\mathsf{PubCited}}[c]$ and then $V_{\mathsf{Text}}[c]$. Finally, take the remaining basic variables in any order. The remaining variables are those whose arguments are researchers or publications that do not exist in ω, along with $V_{\mathsf{NthAuthor}}[(\mathsf{Publication}, i), n]$ variables such that n is greater than $V_{\mathsf{NumAuthors}}[(\mathsf{Publication}, i)](\omega)$.

The aircraft tracking model of Figure 4.4 is not a discrete BLOG model. However, if we define a partition for each basic variable in the same way we did in Definition 4.13, we can construct a supportive numbering for a world ω in this model as follows.

First take N_{Aircraft}. Then do the following for each natural number t in numerical order. Take $V_{\mathsf{State}}\left[(\mathsf{Aircraft}, i), t\right]$ and then $N_{\mathsf{Blip}}\left[\mathsf{Source} = (\mathsf{Aircraft}, i), \mathsf{Time} = t\right]$ for each $i \leq N_{\mathsf{Aircraft}}(\omega)$. Then take $N_{\mathsf{Blip}}\left[\mathsf{Time} = t\right]$. Next, take $V_{\mathsf{MeasuredPos}}\left[b\right]$ for each blip b of the form $(\mathsf{Blip}, (\mathsf{Source}, (\mathsf{Aircraft}, i)), (\mathsf{Time}, t), n)$ where $i \leq N_{\mathsf{Aircraft}}(\omega)$ and $n \leq N_{\mathsf{Blip}}\left[\mathsf{Source} = (\mathsf{Aircraft}, i), \mathsf{Time} = t\right](\omega)$, and each blip b of the form $(\mathsf{Blip}, (\mathsf{Time}, t), n)$ where $n \leq N_{\mathsf{Blip}}\left[\mathsf{Time} = t\right](\omega)$. This numbering covers all basic variables defined on objects that exist in ω. However, we still need to cover the other basic variables; and since the numbering we just described is infinite (there is no upper bound on t), we cannot just add the remaining variables at the end. What we can do is interleave them with the variables already included. Specifically, after the variables defined on existing objects for time t, we can add all basic variables (not included so far) that are defined on time steps $t' \leq t$ and objects of the form $(\mathsf{Aircraft}, i)$, $(\mathsf{Blip}, (\mathsf{Source}, (\mathsf{Aircraft}, i)), (\mathsf{Time}, t''), n)$ or $(\mathsf{Blip}, (\mathsf{Time}, t''))$ where $i, n, t'' \leq t$.

We have shown that in the four BLOG models we have used as examples, every possible world has a supportive numbering. In general, however, determining whether a BLOG model has this property is an undecidable problem.

Proposition 4.8. *The problem of determining whether every possible world of a given BLOG model has a supportive numbering is undecidable.*

Proof. Our proof builds on standard results from first-order logic [Enderton, 2001]. By Church's theorem [1936], the problem of determining whether a given sentence of a first-order logical language is satisfiable (*i.e.*, is true in some model structure) is undecidable. In fact, this remains true if we restrict the first-order language to consist of a single binary predicate symbol [Kalmár, 1936]. We will reduce this problem to that of determining whether every possible world in a BLOG model has a supportive numbering.

Consider any sentence φ of an untyped first-order language $\mathcal{L}$ with a single binary

predicate symbol P. Let M be a BLOG model with one user-defined type Obj, one random function symbol P : (Obj, Obj) → Boolean, and one origin function symbol Parent : Obj → Obj. Let M include one guaranteed object statement, one number statement, and one dependency statement:

```
guaranteed Obj Root;
#Obj(Parent = x) ~ Poisson[1];
P(x1, x2) ~ TabularCPD[[0.5, 0.5]];
```

Thus, every world ω in Ω_M contains at least one object of type Obj, called Root, and for each object o of type Obj there may be some other objects q with $[\mathsf{Parent}]^\omega(q) = o$. So for every natural number $n > 0$, there are worlds $\omega \in \Omega_M$ such that $|\,[\mathsf{Obj}]^\omega\,| = n$; there are also possible worlds where $[\mathsf{Obj}]^\omega$ is countably infinite (note that we are not making any claims about whether such worlds have positive probability). Furthermore, the interpretation of P is unrestricted. So for every model structure of $\mathcal{L}$ containing a finite or countably infinite set of objects, there is model structure in Ω_M that is isomorphic (if one ignores types other than Obj and function symbols other than P).

We can convert the sentence φ into a sentence of the typed language $\mathcal{L}_M$ by letting all quantifiers range over type Obj. Then φ is satisfied by some finite or countably infinite model structure of $\mathcal{L}$ if and only if φ' is satisfied by some world in Ω_M. And by the Löwenheim-Skolem theorem [Löwenheim, 1915; Skolem, 1920], if φ is satisfiable at all, then it is satisfiable in a finite or countably infinite structure.

Now extend M to include two more random Boolean functions with no arguments; call them A and B. Then add the dependency statements:

```
A {
    if φ′ then ~ TabularCPD[[0.1, 0.9],
                            [0.9, 0.1]](B)
```

```
    else ~ TabularCPD[[0.5, 0.5]]
};
B ~ TabularCPD[[0.9, 0.1],
               [0.1, 0.9]](A);
```

Thus in worlds where φ' is false, A depends on nothing and B depends on A. But in worlds where φ' is true, A and B depend on each other, and there is no supportive numbering. So every possible world has a supportive numbering if and only if φ is unsatisfiable. □

This result is not surprising, given that we allow BLOG models to include unrestricted first-order formulas. The absence of an algorithm for seeing if a BLOG model is well-defined does not disqualify BLOG as a representation language: after all, programming languages are useful, and the problem of telling whether a program terminates on all inputs is also undecidable. Section 4.5 gives some criteria that are sufficient to ensure that a BLOG model is well-defined, but these criteria are quite far from being complete enough to identify all well-defined BLOG models.

4.4 Evaluating expressions

As we will see in Chapter 5, one of the basic operations that a BLOG inference algorithm must perform is to determine whether a partial instantiation supports a given basic variable, and if so, to find the applicable elementary CPD and CPD argument values. In order to perform this operation, we need to be able to compute the values of the BLOG expressions that occur in dependency and number statements.

Definition 4.15. *Let e be a BLOG expression, and α be an assignment of values to logical variables such that e is well-formed in* $\mathrm{domain}(\alpha)$*. An achievable instantiation*

σ supports e under α if there is some value q such that for all worlds $\omega \in \mathrm{ev}(\sigma)$, $[e]^{\omega}_{\alpha} = q$. In this case, we say that q is the value of e under α given σ.

In this section, we define a function EVAL-EXPR(e, σ, α) for evaluating expressions given a partial instantiation. If EVAL-EXPR determines that σ does not support e under α, it returns a special value *undet*. This function is incomplete in two senses: it does not always terminate, and it sometimes fails to detect that σ supports e, and thus returns *undet* unnecessarily. In fact, the task we have laid out for EVAL-EXPR is undecidable in general: if φ is a first-order sentence, then a perfect implementation of EVAL-EXPR(φ, $\top$, $\varnothing$) would return false if φ is unsatisfiable, true if $\neg\varphi$ is unsatisfiable, and *undet* if both φ and $\neg\varphi$ are satisfiable. At the end of this section we will show that there is a large class of BLOG expressions (including all those that appear in our running examples) on which EVAL-EXPR is at least guaranteed to terminate. However, even on these expressions, EVAL-EXPR may return *undet* unnecessarily.

4.4.1 Main evaluation functions

Rather than writing EVAL-EXPR as a single giant function, we will break it down into a hierarchy of subroutines. Our EVAL-EXPR function is shown in Figure 4.5; it is just a dispatcher that calls EVAL-FORMULA, EVAL-TERM, or EVAL-SET-EXPR, depending on the type of expression passed in.

Figure 4.6 shows the function EVAL-TERM, which is also fairly straightforward. If the term is a logical variable, its value is given by the assignment that is passed in. If the term is an application of a function symbol to a tuple of arguments (an empty tuple if we are dealing with a constant symbol), then EVAL-TERM calls itself recursively to evaluate each argument term, and then calls GET-FUNC-VALUE to evaluate the function. This recursion terminates at logical variables and constant symbols.

```
function EVAL-EXPR(e, σ, α)
   returns an object, a finite set or multiset, null, or undet
   inputs: e, a BLOG expression
           σ, a finite instantiation of basic random variables
           α, an assignment to the free logical variables in e

   if e is a term
      return EVAL-TERM(e, σ, α)
   if e is a formula
      return EVAL-FORMULA(e, σ, α)
   if e is a set expression
      return EVAL-SET-EXPR(e, σ, α)
```

Figure 4.5: The top-level dispatcher function for evaluating expressions.

Note that EVAL-TERM maintains a distinction between *undet*, which indicates that σ does not determine the denotation of a term, and null, which can actually serve as the denotation of a term. The function GET-FUNC-VALUE contains separate code for getting the values of nonrandom functions, origin functions, and random functions. For random functions, it returns the value obtained by passing a function application variable to GET-VAR-VALUE; the returned value is *undet* if σ does not instantiate this variable.

The function EVAL-FORMULA is shown in Figure 4.7. This function includes straightforward code for handling simple atomic formulas, equality formulas, negations, and conjunctions. The only thing to note in this part of the function is the short-circuit evaluation for conjunctions: if the first conjunct is false, we do not try to evaluate the second conjunct. This reduces the set of inputs on which EVAL-FORMULA returns *undet*.

The most interesting case in EVAL-FORMULA is for existentially quantified formulas (universally quantified formulas are handled using the equivalence between $\forall\ \tau\ x\ \psi$ and $\neg\exists\ \tau\ x\ \neg\psi$). Here we have to iterate over all objects that could possibly satisfy the subformula ψ when bound to x. This is done using an itera-

```
function EVAL-TERM(t, σ, α)
   returns an object or null, or undet
   inputs: t, a logical term
           σ, a finite instantiation of basic random variables
           α, an assignment to the logical variables in t

   if t is a logical variable x
      return α(x)
   if t is a function application term f(t_1, ..., t_k), possibly with k = 0
      (o_1, ..., o_k) ← a new array of length k
      for i = 1 to k do
         o_i ← EVAL-TERM(t_i, σ, α)
         if o_i = undet return undet
         if o_i = null return null
      return GET-FUNC-VALUE(f, (o_1, ..., o_k), σ)
```

```
function GET-FUNC-VALUE(f, (o_1, ..., o_k), σ)
   returns an object or null, or undet
   inputs: f, a function symbol
           (o_1, ..., o_k), a tuple of objects (possibly empty)
           σ, a finite instantiation on the basic random variables

   if f is a nonrandom function symbol
      interp ← GET-NONRANDOM-INTERP(f)
      return interp(o_1, ..., o_k)
   if f is an origin function symbol
      if o_1 is a non-guaranteed object of the form (..., (f, o), ...)
         return o
      return null
   if f is a random function symbol
      return GET-VAR-VALUE(σ, V_f [o_1, ..., o_k])
```

Figure 4.6: Functions for evaluating logical terms.

function EVAL-FORMULA(φ, σ, α)
  returns a truth value or *undet*
  inputs: φ, a logical formula
    σ, a finite instantiation of basic random variables
    α, an assignment to the free variables in φ

  if φ is a Boolean term t
    $val \leftarrow$ EVAL-TERM(t, σ, α); **if** val = undet **return** undet
    if val = true **return** true
    return false

  if φ has the form $t_1 = t_2$
    $val_1 \leftarrow$ EVAL-TERM(t_1, σ, α); **if** val_1 = undet **return** undet
    $val_2 \leftarrow$ EVAL-TERM(t_2, σ, α); **if** val_2 = undet **return** undet
    if $val_1 = val_2$ **return** true
    return false

  if φ has the form $\neg\ \psi$
    $val \leftarrow$ EVAL-FORMULA(ψ, σ, α); **if** val = undet **return** undet
    if val = true **return** false
    return true

  if φ has the form $\psi\ \wedge\ \chi$
    $val_1 \leftarrow$ EVAL-FORMULA(ψ, σ, α); **if** val_1 = undet **return** undet
    if val_1 = false **return** false
    $val_2 \leftarrow$ EVAL-FORMULA(χ, σ, α); **if** val_2 = undet **return** undet
    if val_2 = false **return** false
    return true

  if φ has the form $\exists\ \tau\ x\ \psi$
    $iter \leftarrow$ GET-POTENTIAL-SATISFIERS-ITER(τ, x, ψ, $\{x\}$, σ, α)
    while CAN-DETERMINE-NEXT($iter$) and HAS-NEXT($iter$) **do**
      $o \leftarrow$ GET-NEXT($iter$)
      $\alpha' \leftarrow (\alpha;\ (x,\ \tau)\ \mapsto\ o)$
      $val \leftarrow$ EVAL-FORMULA(ψ, σ, α'); **if** val = undet **return** undet
      if val = true **return** true
    if not CAN-DETERMINE-NEXT($iter$) **return** undet
    return false

  if φ has the form $t_1 \neq t_2$ **return** EVAL-FORMULA($\neg(t_1 = t_2)$, σ, α)
  if φ has the form $\psi\ \vee\ \chi$ **return** EVAL-FORMULA($\neg(\neg\ \psi\ \wedge\ \neg\ \chi)$, σ, α)
  if φ has the form $\psi\ \rightarrow\ \chi$ **return** EVAL-FORMULA($\neg(\psi\ \wedge\ \neg\ \chi)$, σ, α)
  if φ has the form $\forall\ \tau\ x\ \psi$ **return** EVAL-FORMULA($\neg\ \exists\ \tau\ x\ \neg\ \psi$, σ, α)

Figure 4.7: Function for evaluating logical formulas.

tor returned by GET-POTENTIAL-SATISFIERS-ITER. A perfect implementation of GET-POTENTIAL-SATISFIERS-ITER would return an iterator over exactly those objects that satisfy ψ when bound to x, if this set is the same in all worlds consistent with σ. However, the implementation that we will describe in Section 4.4.2 is not so precise. The returned iterator may range over a larger set, although it is limited to objects that exist in all worlds in $\text{ev}(\sigma)$.

The function CAN-DETERMINE-NEXT(*iter*) tries to determine what object the iterator *iter* should return next. This may involve looking at the values of some random variables in σ; CAN-DETERMINE-NEXT returns true just when all the variables it tries to look at are instantiated in σ. The function HAS-NEXT returns true if the iterator has another object to return, and GET-NEXT returns the next object that has not been returned so far. Because the objects returned by the iterator may or may not actually satisfy ψ, EVAL-FORMULA checks each object individually using a recursive call on ψ. If CAN-DETERMINE-NEXT ever returns false, EVAL-FORMULA returns *undet*. If the end of the iteration is reached without any objects satisfying ψ, EVAL-FORMULA returns false.

Finally, the last few lines of EVAL-FORMULA handle the remaining types of formulas by reducing them to the ones handled by the earlier cases. Inequality formulas are reduced to negated equalities; disjunctions and implications are reduced to negated conjunctions (with one or both of the sub-formulas negated), and universal formulas are reduced to negated existential formulas.

We still have to specify how to evaluate set expressions: this is done by the function EVAL-SET-EXPR shown in Figure 4.8. An explicit set expression can be evaluated simply by evaluating each term that it contains. For an implicit set expression, the code is similar to that for an existentially quantified formula. The main difference is that with the existential quantifier, we could short-circuit when we found one object satisfying the subformula ψ; with the set expression, we have

```
function EVAL-SET-EXPR(e, σ, α)
  returns a finite set or multiset, a natural number, or undet
  inputs: e, a set expression
          σ, a finite instantiation of basic random variables
          α, an assignment to the free variables in e

  if e is an explicit set expression {t_1, ..., t_k}
    S ← an empty set
    for i = 1 to k do
      o ← EVAL-TERM(t_i, σ, α)
      if o = undet return undet
      ADD-TO-SET(S, o)
    return S

  if e is an implicit set expression {τ x : φ}
    S ← an empty set
    iter ← GET-POTENTIAL-SATISFIERS-ITER(τ, x, φ, {x}, σ, α)
    while CAN-DETERMINE-NEXT(iter) and HAS-NEXT(iter) do
      o ← GET-NEXT(iter)
      α' ← (α; (x, τ) ↦ o)
      val ← EVAL-FORMULA(φ, σ, α')
      if val = undet return undet
      if val = true
        ADD-TO-SET(S, o)
    if not CAN-DETERMINE-NEXT(iter) return undet
    return S

  if e is a tuple multiset expression {t_1, ..., t_k for τ_1 x_1, ..., τ_n x_n : φ}
    return EVAL-TUPLE-MULTISET-EXPR
             ((t_1, ..., t_k), ((x_1, τ_1), ..., (x_n, τ_n)), φ, σ, α)

  if e is a cardinality expression #s
    S ← EVAL-SET-EXPR(s, σ, α)
    if S = undet return undet
    return |S|
```

Figure 4.8: Function for evaluating set expressions.

to accumulate all satisfiers in a set. The same idea applies to tuple multiset expressions, although the implementation is more complicated. We use a recursive function Eval-Tuple-Multiset-Expr, shown in Figure 4.9. The insight behind this function is that the assignments to $x_1, \ldots, x_n$ that satisfy φ can be partitioned according to the value that they assign to x_1. For each possible value of x_1, we can evaluate a tuple multiset expression over $x_2, \ldots, x_n$, and so on. The base case is when the tuple multiset expression does not range over any variables; then the assignment α can be used to evaluate the condition φ and the terms $t_1, \ldots, t_k$.

Note that in Eval-Tuple-Multiset-Spec, we end up calling Get-Potential-Satisfiers-Iter$(\tau, x, \varphi, F, \sigma, \alpha)$ with a set of free variables F that includes not just x, but also other logical variables that may occur in φ and are not included in α. This means that the iterator must return all values for x that satisfy φ with *some* assignment of values to the remaining variables in F.

4.4.2 Enumerating objects that may satisfy a formula

We will now examine how to implement the function Get-Potential-Satisfiers-Iter, which returns an iterator over objects that may satisfy a formula φ when bound to a logical variable x in the worlds consistent with an instantiation σ. One of the arguments to Get-Potential-Satisfiers-Iter is a tuple $\mathbf{b}$ of variable-type pairs, which includes a type τ for x. Thus, one simple way to implement the function is to return an iterator over all objects of type τ. If σ does not determine which non-guaranteed objects of type τ exist, then Can-Determine-Next will return false on the iterator at some point.

This implementation is fine for evaluating some of the set expressions we have used in our examples, such as {`Ball b`} in the urn-and-balls model and {`Publication p`} in the citations model. In these cases, the set of objects of the relevant type is de-

```
function EVAL-TUPLE-MULTISET-EXPR(t, b, φ, σ, α)
    returns a finite multiset of tuples, or undet
    inputs: t, a tuple of terms (t_1, ..., t_k)
            b, a tuple of variable-type pairs
            φ, a formula
            σ, a finite instantiation of basic random variables
            α, an assignment to the free variables in t and φ that are not in b

    if b is empty
        val ← EVAL-FORMULA(φ, σ, α); if val = undet return undet
        if val = true
            (o_1, ..., o_k) ← a new array of length k
            for i = 1 to k do
                o_i ← EVAL-TERM(t_i, σ, α); if val = undet return undet
            return a multiset consisting of the single tuple (o_1, ..., o_k)
        return an empty multiset

    (x, τ) ← FIRST(b)
    b' ← REST(b)
    F ← the set of variables in b
    M ← an empty multiset
    iter ← GET-POTENTIAL-SATISFIERS-ITER(τ, x, φ, F, σ, α)
    while CAN-DETERMINE-NEXT(iter) and HAS-NEXT(iter) do
        o ← GET-NEXT(iter)
        α' ← (α; x ↦ o)
        N ← EVAL-TUPLE-MULTISET-EXPR(t, b', φ, σ, α'); if N = undet return undet
        MERGE-INTO-MULTISET(M, N)
    if not CAN-DETERMINE-NEXT(iter) return undet
    return M
```

Figure 4.9: Function for evaluating tuple multiset expressions.

termined by a single number variable. If σ does not instantiate this number variable, then the first call to CAN-DETERMINE-NEXT on the iterator returns false; otherwise, the iterator ranges over a finite set, because the number of objects satisfying a given number variable is finite in each possible world (by Definition 4.10(iv)).

However, consider the tuple multiset expression that we use in the citations model in Figure 4.3:

```
{n, Name(NthAuthor(PubCited(c), n))
                    for NaturalNum n : n < NumAuthors(PubCited(c))}
```

To evaluate this expression, it is not sufficient to iterate over all natural numbers and check whether each one is less than the value denoted by NumAuthors(PubCited(c)). The problem is that although the iterator *iter* may eventually return all the natural numbers less than this upper bound, EVAL-TUPLE-MULTISET-EXPR does not stop iterating until HAS-NEXT(*iter*) returns false — which will never happen if the iterator ranges over all natural numbers.

Another case where the naive implementation of GET-POTENTIAL-SATISFIERS-ITER falls short is in the aircraft tracking model, if we introduce random constant symbols to refer to the blips that we observe at certain times (this idea is discussed further in Section 5.1.3). Suppose we take the model in Figure 4.4 and add a new dependency statement:

```
random Blip Blip1 ~ Uniform({Blip b : (Time(b) = 8)});
```

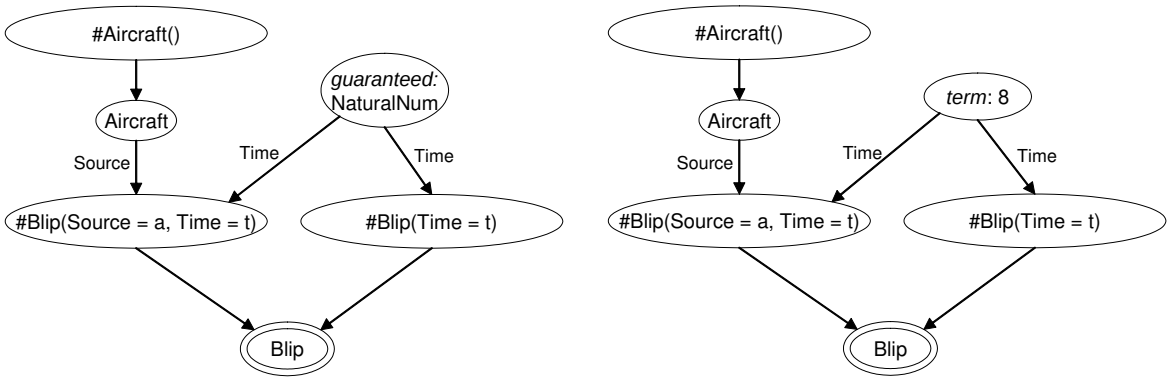
This statement asserts that Blip1 denotes some blip the appears at time 8 (or null in worlds where no such blip exists). It uses an implicit set expression containing those blips that, when bound to the logical variable b, satisfy the formula Time(b) = 8. However, we cannot evaluate this expression just by iterating over all blips that

exist in a world. Since the aircraft model includes an infinite sequence of time steps, the full set of blips in a possible world is determined by an infinite sequence of number variables. A finite instantiation σ cannot instantiate all of these variables; thus, if we try to evaluate this set expression using an iterator over all radar blips, EVAL-SET-EXPR will always end up returning *undet*. This is unnecessary, because the number of blips at time 8 is governed by one number variable per aircraft (plus one variable for false alarm blips) and the number of aircraft in each world is finite.

4.4.3 Object generation graphs

To handle cases like those described above, we need to use iterators that range over more tightly defined sets. Specifically, we would like to constrain the magnitudes of natural numbers, and constrain the values of origin functions (such as Time) on non-guaranteed objects. Such constrained sets of objects can be represented by *object generation graphs* (OGGs). Semantically, an OGG behaves like a set expression: given a world ω and a logical variable assignment α, an OGG G denotes a set of objects $[G]^{\omega}_{\alpha}$. The objects in this set are said to *satisfy* G in ω given α. The point of representing a constrained set of objects with an OGG, rather than with a standard set expression, is that we have an algorithm for iterating over exactly those objects that satisfy a given OGG; we have no such algorithm for set expressions in general.

As an example, consider the object generation graphs in Figure 4.10. In any given world ω, the graph in part (a) represents the set of all objects of type Blip that exist in ω. It contains three kinds of nodes: *type nodes* labeled with the types Blip and Aircraft; *number statement nodes* labeled with the headers of number statements; and a *guaranteed object node* for type NaturalNum. The graph in part (b), on the other hand, represents a constrained set of blips: those on which the origin function Time returns 8. This graph contains an additional kind of node: a *term node* labeled



(a) {`Blip b`} (b) {`Blip b: Time(b) = 8`}

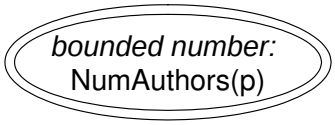
Figure 4.10: Object generation graphs for two set expressions involving radar blips.

with a logical term, in this case the constant "8".

Each OGG contains a distinguished *target node*, shown with a double oval in our diagrams. The objects that satisfy the OGG are those that satisfy this target node, in a sense that we will make precise below. Intuitively, the graph specifies the ways in which objects satisfying the target node can be generated. In Figure 4.10(a), blips can satisfy one of two number statements: one with origin functions Source and Time, and one with a single origin function Time (for false detections). The aircraft that serve as values for Source are, in turn, generated by a number statement for aircraft. The value of Time can be any guaranteed object of type NaturalNum. The difference in Figure 4.10(b) is that only one natural number can serve as a value for Time, namely the number denoted by the term "8".

There is one more kind of node that can occur in OGGs: *bounded number nodes*, which represent subsets of the natural numbers upper-bounded by the values of certain terms. Such a node can be used, for example, to define the set of satisfiers for {`NaturalNum n : n < NumAuthors(p)`}, as shown in Figure 4.11(b). In this figure, the bounded number node constitutes the whole OGG on its own, but such

(a) {NaturalNum n}



(b) {NaturalNum n :
n <
NumAuthors(p)}

Figure 4.11: Single-node object generation graphs for two set expressions involving natural numbers.

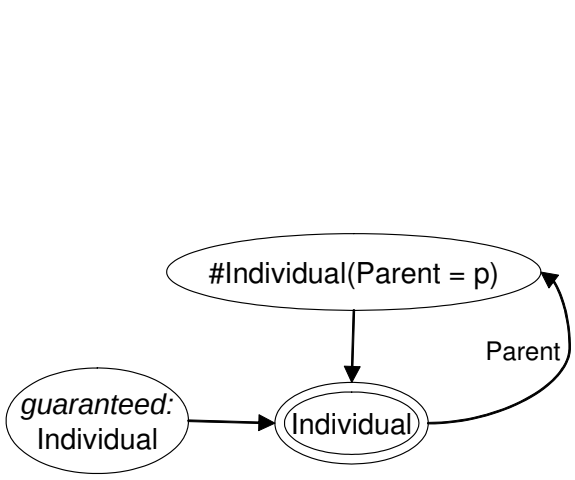
nodes can also be used in larger graphs. For example, we could replace the term node in Figure 4.10(b) with a bounded number node to define the set of nodes whose Time value is less than 8. In general, a bounded number node can include a set of terms that serve as upper bounds: then a natural number satisfies the node if it is less than the denotations of all the listed terms. This allows us to represent, for example, {NaturalNum n : n < NumAuthors(p) & n < 2}. In some worlds and under some logical variable assignments, NumAuthors(p) may provide a tighter bound than "2", whereas "2" may provide the tighter bound in other contexts. We adopt the convention that if a term denotes null, then no number is less than it.[2]

In models with recursive number statements, OGGs may contain cycles. For instance, consider the following BLOG model for asexual reproduction:

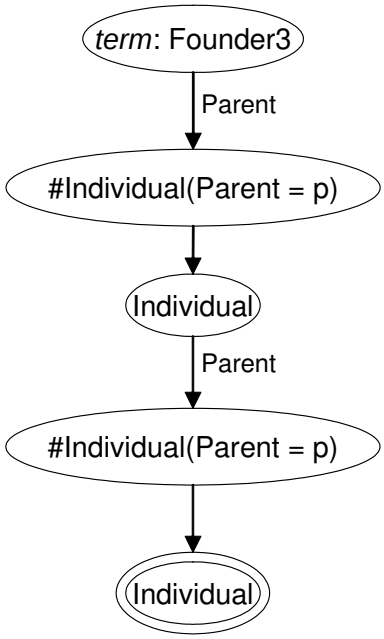
```
type Individual;
origin Individual Parent(Individual);
guaranteed Individual Founder1, Founder2, Founder3;
#Individual(Parent = i) ~ NumChildrenPrior;
```

In this model, individuals can generate other individuals *ad infinitum*. Figure 4.12(a)

[2]This is consistent with the fact that the formula $n <$ null is always false — which is the case because the term LessThan(n, null) denotes null, and an atomic formula is satisfied just when the underlying term denotes true (see Definition 2.8).



(a) {`Individual i`}



(b) {`Individual i :`
`Parent(Parent(i))`
`= Founder3`}

Figure 4.12: Object generation graphs for two set expressions in a model of asexual reproduction.

shows an OGG that has the same denotation as the unconstrained set expression {`Individual i`}. An individual can be a guaranteed individual, or it can be generated by a parent; that parent, in turn, can either be guaranteed or have a parent, and so on. Thus, this graph is cyclic. Figure 4.12(b) shows an acyclic OGG denoting the set of individuals that are "grandchildren" of Founder3. This graph illustrates that an OGG may contain several type nodes labeled with the same type, and several number statement nodes labeled with the same number statement. Because these nodes have different parents, they are satisfied by different objects: in the figure, the top Individual node is satisfied by children of Founder3, and the bottom one is satisfied by the grandchildren of Founder3.

We are now ready to give a formal definition of an OGG.

Definition 4.16. *In a BLOG model M, an* object generation graph *(OGG) is a finite directed graph in which each node may be of one of the following five kinds:*

- *a* type node*, labeled with a user-defined type symbol from M;*
- *a* number statement node*, labeled with the header of a number statement from M;*
- *a* guaranteed object node*, labeled with the keyword* guaranteed *and a type symbol from M;*
- *a* term node*, labeled with a term of $\mathcal{L}_M$;*
- *a* bounded number node*, labeled with the keyword* boundednumber *and a set of terms of $\mathcal{L}_M$.*

The parents of a type node labeled with τ must be number statement nodes or guaranteed object nodes for type τ. Edges into number statement nodes are labeled with origin functions; a number statement with origin functions $g_1, \ldots, g_k$ must have exactly one incoming edge labeled with each of these functions. Also, exactly one node is marked as the target node *for the OGG.*

An OGG G is *well-formed* in a scope β if every term in G is well-formed in that scope and all the terms used by bounded number nodes have type NaturalNum. We will now define what it means for an object to satisfy a node in an OGG.

Definition 4.17. *Let G be an OGG for a BLOG model M, ω be a possible world of M, and α be an assignment that is valid in ω, such that G is well-formed in* $\text{domain}(\alpha)$*. Let u be a node in G, and o be an object in $\mathcal{U}_M(\tau)$ for some type τ. Then o* satisfies *u in ω under α if one of the following conditions holds:*

- *u is a type node, and o satisfies one of the parents of u;*
- *u is a number statement node for type τ with origin functions $g_1, \ldots, g_k$, o is a non-guaranteed object in $[\tau]^\omega$ that has the form $(\tau, (g_1, o_1), \ldots, (g_k, o_k), n)$, and for each $i \in \{1, \ldots, k\}$, o_i satisfies the parent of u whose edge into u is labeled with g_i;*
- *u is a guaranteed object node for type τ and $o \in \mathrm{Guar}_M(\tau)$;*
- *u is a term node labeled with a term t and $o = [t]^\omega_\alpha$;*
- *u is a bounded number node labeled with terms $\{t_i\}_{i=1}^k$ and o is a natural number that is less than $[t_i]^\omega_\alpha$ for all $i \in \{1, \ldots, k\}$ (where we consider no natural number to be less than* null*).*

The denotation *of a node u in ω under α, written $[u]^\omega_\alpha$, is the set of objects that satisfy u in ω under α. The denotation of G is the denotation of its target node.*

Two of the cases in this definition are recursive: o satisfies a type node if it satisfies one of the node's parents, and o satisfies a number statement node if its generating objects satisfy the node's parents. To see that this recursion is well-founded even for cyclic OGGs, note that we can define the satisfaction relation first for guaranteed objects, then for non-guaranteed objects whose tuple representations only have one level of nesting, then for non-guaranteed objects with two levels of nesting, etc. By Definition 4.9, each tuple in $\mathcal{U}_M(\tau)$ is nested to some finite depth. When defining the satisfaction relation for a given object o, we can begin by defining the conditions under which it satisfies number statement nodes and guaranteed object nodes. These are the only kinds of nodes that can be parents of type nodes, so next we can define when o satisfies type nodes. Finally, we can handle term nodes and bounded number nodes. Thus, our recursive definition is well-formed.

The following lemma gives conditions under which the set of objects satisfying an OGG is finite.

Lemma 4.9. *In a BLOG model M, if G is an object generation graph that is acyclic and does not contain guaranteed object nodes for any infinite built-in types, then $[G]^\omega_\alpha$ is finite for every world $\omega \in \Omega_M$ and every assignment α that is valid in ω.*

Proof. The definition of an OGG stipulates that G must be a finite graph, so if G is acyclic, then it has a topological numbering. We can show by induction on this numbering that each node has finitely many satisfiers. Consider any node u in this numbering, and assume that each of its predecessors has finitely many satisfiers. If u is a term node, then it has just one satisfier (or zero if the term evaluates to null in ω under α). If u is a guaranteed object node, then it must be for a finite built-in type (namely Boolean) or for a user-defined type, which can only have finitely many guaranteed objects. If u is a bounded number node, then it has at least one upper-bound term t, which evaluates to a finite number (or null) in ω under α. In this case, the set of satisfiers of u has size at most $[t]^\omega_\alpha$, or zero if the term denotes null.

Now suppose u is a number statement node for type τ with origin functions $g_1, \ldots, g_k$ and corresponding parents $v_1, \ldots, v_k$. Then the objects that satisfy u are in $[\tau]^\omega$ and have the form $(\tau, (g_1, o_1), \ldots, (g_k, o_k), n)$, where $(o_1, \ldots, o_k) \in [v_1]^\omega_\alpha \times \cdots \times [v_k]^\omega_\alpha$. Since $v_1, \ldots, v_k$ are all predecessors of u, we know they have finite satisfier sets. Thus the number of tuples $(o_1, \ldots, o_k)$ in $[v_1]^\omega_\alpha \times \cdots \times [v_k]^\omega_\alpha$ is finite. For each such tuple, the number of generated objects $(\tau, (g_1, o_1), \ldots, (g_k, o_k), n)$ in $[\tau]^\omega$ is finite by Definition 4.10(iv). So u has finitely many satisfiers.

Finally, if u is a type node, then its satisfier set is the union of its parents' satisfier sets. Since its parents' satisfier sets are finite, u's satisfier set is finite as well. □

This lemma implies that the OGG for {`Blip b: Time(b) = 8`}, shown in Figure 4.10(b), has finitely many satisifiers. The same is true of the OGG for {`NaturalNum`

`n: n < NumAuthors(p)`}, shown in Figure 4.11(b), and the OGG for {`Individual i : Parent(Parent(i)) = Founder3`}, shown in Figure 4.12(b). The lemma does not apply to the graphs in part (a) of these Figures: the graphs in Figure 4.10(a) and 4.11(a) contain guaranteed object nodes for the infinite type NaturalNum, and the graph in Figure 4.12(a) contains a cycle.

4.4.4 Constructing object generation graphs for a formula

We now discuss an algorithm for constructing OGGs automatically: for example, taking the set expression {`Blip b: Time(b) = 8`} and constructing the OGG in Figure 4.10(b). Recall that our purpose in using OGGs is to enumerate all the objects that might satisfy a formula when bound to a certain variable. When we actually enumerate such objects in Eval-Expr and its subroutines, we are working in the context of a particular instantiation σ and logical variable assignment α. However, our procedure does not take any particular σ or α into account: it constructs OGGs that can be used under all instantiations and assignments. This means that we can construct OGGs for the formulas and set expressions in a BLOG model in a preprocessing step, before beginning to evaluate expressiosn or run an inference algorithm. Our procedure for constructing OGGs, called Construct-OGGs, is shown in Figure 4.13. We show at the end of this section that Construct-OGGs is sound, in that we will not miss any formula satisfiers if we iterate over all the satisfiers of the constructed OGGs.

In general, Construct-OGGs returns not just one OGG, but a list of them. This is useful for representing the satisfier sets of disjunctive expressions, such as {`Blip b: Time(b) = 3 | Time(b) = 8`}: we end up with one OGG per disjunct. The arguments to Construct-OGGs are the satisfier type τ, the variable x to which potential satisfiers are bound, the formula φ to be satisfied, and a set F of free

variables that includes x. The constructed OGGs should not use terms containing any of these free variables. As an example of a case where F consists of more than one variable, consider the following tuple multiset expression in our asexual reproduction model (with an additional function symbol Distance):

```
{Distance(i, j) for Individual i, Individual j : Parent(i) = j}
```

When evaluating this expression, Eval-Tuple-Multiset-Expr iterates over bindings to i in an outer loop, and for each possible value of i, it iterates over possible bindings to j. Thus, the OGG used for iterating over i must treat both i and j as free variables: neither of these variables has a fixed value throughout the iteration. On the other hand, the OGG for iterating over j can treat i as a bound variable.

As we can see in Figure 4.13, the first thing that Construct-OGGs does is convert the given formula φ to *disjunctive normal form* (DNF): that is, a disjunction of conjunctions (see Enderton [2001]). In general, converting a first-order formula to disjunctive normal form involves moving all quantifiers to the beginning of the formula, and replacing existentially quantified variables with Skolem constants. However, we use a "shallow" version of DNF in which we do not modify quantified sub-formulas of φ. Thus, the conjunctions in the resulting DNF formula can contain both *literals* — that is, atomic formulas and equality formulas, possibly negated — and quantified sub-formulas. The rest of our OGG construction algorithm simply ignores the constraints imposed by the quantified subformulas; this does not jeopardize the soundness of the algorithm, because ignoring certain constraints only enlarges the set of objects that satisfy an OGG. Because of this shallow conversion, the function Get-Shallow-DNF can just use a version of the algorithm for converting a propositional formula to DNF.

Construct-OGGs begins with an empty graph for each disjunct in the DNF

```
function CONSTRUCT-OGGS(τ, x, φ, F)
   returns a list of object generation graphs
   inputs: τ, a type
           x, a logical variable
           φ, a formula
           F, a set of logical variables

   (c_1, ..., c_m) ← GET-SHALLOW-DNF(φ)
   (G_1, ..., G_m) ← a list of m empty object generation graphs
   for i = 1 to m do
      L ← SELECT-LITERALS-CONTAINING(c_i, x)
      u ← GET-OR-ADD-OGG-NODE(G_i, τ, x, L, F)
      SET-TARGET-NODE(G_i, u)
   return (G_1, ..., G_m)
```

```
function GET-OR-ADD-OGG-NODE(G, τ, t, L, F)
   returns a node in G, possibly added to G by this function
   inputs: G, an object generation graph
           τ, a type
           t, a term
           L, a list of literals, each of which contains t
           F, a set of logical variables

   for each literal ℓ in L do
      if ℓ has the form t = t' where t' does not contain any variables in F
         return GET-OR-ADD-TERM-NODE(G, t')

   if τ is built-in
      if τ = NaturalNum
         bounds ← empty set
         for each literal ℓ in L do
            if ℓ has the form t < t' where t' does not contain any variables in F
               ADD-TO-SET(bounds, t')
         if bounds is not empty
            return GET-OR-ADD-BOUNDED-NUMBER-NODE(G, bounds)
      return GET-OR-ADD-GUARANTEED-OBJ-NODE(G, τ)
   return GET-OR-ADD-TYPE-NODE(G, τ, t, L, F)
```

Figure 4.13: Functions used to construct an object generation graph for a variable in a formula.

formula. Then for each disjunct, it calls GET-OR-ADD-OGG-NODE to add the desired target node to the graph. As a side effect, this function adds any ancestors of the target node that need to be included. Pseudocode for GET-OR-ADD-OGG-NODE is also given in Figure 4.13. In general, this function returns a node whose satisfiers include all the possible values of a term t given the constraints imposed by the literals $\mathbf{L}$; for now, t will simply be the variable x. The function begins by checking whether any of the literals asserts that t is equal to a particular term t' (containing none of the free variables in F). If there is such a literal, then GET-OR-ADD-OGG-NODE returns a term node for t', adding it to the graph if it does not already exist. Otherwise, the kind of node returned depends on the type τ. If τ is NaturalNum, then the function scans the literals for upper bounds on t. If it finds some, then it returns a bounded number node. Otherwise, it reverts to its default behavior for built-in types, which is to return a guaranteed object node. If τ is not built-in, then the function returns a type node with certain parents; this node is constructed by the function GET-OR-ADD-TYPE-NODE.

Figure 4.14 gives pseudocode for GET-OR-ADD-TYPE-NODE (the other GET-OR-ADD functions are trivial). Recall that an OGG may contain several type nodes for the same type, having different parents and thus different satisfier sets. However, if GET-OR-ADD-TYPE-NODE is invoked with an empty list of literals $\mathbf{L}$, then the satisfier set of the resulting type node will be all objects of that type; there is no point in creating multiple such nodes. Thus, the OGG data structure keeps track of an unconstrained type node for each user-defined type: this node is accessed with GET-UNCONSTRAINED-TYPE-NODE and set with SET-UNCONSTRAINED-TYPE-NODE. If $\mathbf{L}$ is empty and an unconstrained type node already exists for type τ, then GET-OR-ADD-TYPE-NODE simply returns that node. Otherwise, it creates a new node, and sets it as the unconstrained type node for type τ if appropriate.

The next order of business is to determine the parents of the new type node.

```
function GET-OR-ADD-TYPE-NODE(G, τ, t, L, F)
  returns a user-defined type node in G, possibly added by this function
  inputs: G, an object generation graph
          τ, a type
          t, a term
          L, a list of literals, each of which contains t
          F, a set of logical variables

  if L is empty
    u ← GET-UNCONSTRAINED-TYPE-NODE(G, τ)
    if u ≠ null return u
  u ← ADD-TYPE-NODE(G, τ)
  if L is empty
    SET-UNCONSTRAINED-TYPE-NODE(G, τ, u)

  nonNullFuncs ← an empty set
  for each origin function g for type τ do
    for each literal ℓ in L do
      t' ← MAKE-TERM(g, (t))
      if (ℓ has the form t_1 ≠ null where t_1 contains t')
            or (ℓ has the form t_1 = t_2 where t_1 contains t'
                and t_2 is a variable or a nonrandom constant not denoting null)
          ADD-TO-SET(nonNullFuncs, g)

  if nonNullFuncs is empty and Guar_M(τ) is not empty
    v ← GET-OR-ADD-GUARANTEED-OBJ-NODE(G, τ)
    ADD-EDGE(G, v, u)
  for each number statement s for type τ do
    if the origin functions in s include all of nonNullFuncs
      v ← ADD-NUM-STMT-NODE(G, s)
      ADD-EDGE(G, v, u)
      for each origin function g in s do
        t' ← MAKE-TERM(g, (t))
        L' ← SELECT-LITERALS-CONTAINING(L, t')
        w ← GET-OR-ADD-OGG-NODE(G, ret(g), t', L', F)
        ADD-EDGE-WITH-LABEL(G, w, v, g)
  return u
```

Figure 4.14: Function that constructs a user-defined type node in an OGG.

Recall that in general, the parents of a type node can include a guaranteed object node and number statement nodes for that type. However, if $\mathbf{L}$ contains literals asserting that certain origin functions yield non-null values on t, then we can limit the set of parent nodes while still preserving soundness. If an origin function g yields a non-null value on t, then the value of t cannot be a guaranteed object, and must satisfy a number statement that uses g. Thus, GET-OR-ADD-TYPE-NODE constructs a set of origin functions called *nonNullFuncs*. Each origin function g is added to *nonNullFuncs* if some literal implies that the term $t' = g(t)$ has a non-null value. There are several ways that a literal can imply this. First, since all function application terms denote null when one of their arguments denotes null (Definition 2.7), it suffices to imply that some term t_1 containing t' has a non-null value. This can be implied by the formula $t_1 \neq \mathsf{null}$, or by a formula $t_1 = t_2$ where t_2 is a term that cannot denote null in any world under any valid assignment — namely a logical variable, or a nonrandom constant symbol denoting a non-null value. For example, the literal $\mathsf{Time}(b) = 8$ implies that Time has a non-null value on b, and the literal $\mathsf{Parent}(\mathsf{Parent}(i)) = \mathsf{Founder3}$ implies that Parent has a non-null value on both i and $\mathsf{Parent}(i)$.

Once the set of non-null origin functions is determined, parents can be added to the type node. If *nonNullFuncs* is empty, then t may denote a guaranteed object, so a guaranteed object node is added as a parent. A parent node is also added for each number statement that includes all the necessary origin functions. Number statement nodes may also need parents, one for each origin function. The parent corresponding to origin function g is obtained using a recursive call to GET-OR-ADD-OGG-NODE, passing in $\mathrm{ret}(g)$ as the type and $g(t)$ as the term whose possible values must be represented. The set of literals passed in is limited to those that contain $g(t)$. To see that this recursion eventually terminates, note that if the depth of recursion exceeds the maximum depth of nested terms in the original formula φ passed to

CONSTRUCT-OGGs, the set of literals will be empty. At this point, any type nodes that are created will serve as the unconstrained nodes for their types. There can be at most one additional call to GET-OR-ADD-OGG-NODE for each type before GET-OR-ADD-TYPE-NODE starts returning unconstrained nodes that already exist. Thus, regardless of the structure of the BLOG model, CONSTRUCT-OGGs always terminates and returns a value.

We can now define the sense in which CONSTRUCT-OGGs is correct: the returned list of OGGs *covers* the variable x in the formula φ.

Definition 4.18. *In a BLOG model M, let x be a logical variable, φ be a formula, and β_F be a scope that assigns types to x and possibly to some other variables. A list of object generation graphs $G_1, \ldots, G_m$* covers *x in φ with free-variable scope β_F if the following condition holds. Let ω be any world in Ω_M and α be any assignment that is valid in ω, such that φ is well-formed in the scope* $(\text{domain}(\alpha); \beta_F)$*. Then $G_1, \ldots, G_m$ are well-formed in* $\text{domain}(\alpha)$*, and for any assignment α_F to β_F that is valid in ω,*

$$\omega \models_{(\alpha;\alpha_F)} \varphi \quad \textit{implies} \quad \alpha_F(x) \in \bigcup_{i=1}^{m} [G_i]_\alpha^\omega$$

To understand this definition better, consider the simple case where β_F consists of a single variable-type pair (x, τ). Then the condition states that for any $o \in [\tau]^\omega$,

$$\omega \models_{(\alpha;(x,\tau)\mapsto o)} \varphi \quad \text{implies} \quad o \in \bigcup_{i=1}^{m} [G_m]_\alpha^\omega$$

In other words, if φ is true in ω under α with x bound to o, then o is in the union of the denotations of $G_1, \ldots, G_m$.

Lemma 4.10. *Suppose* CONSTRUCT-OGGS*(τ, x, φ, F) returns a list of OGGs $G_1, \ldots, G_m$. Let β_F be any scope that assigns types to the variables in F such that $\beta_F(x) = \tau$. Then the list $G_1, \ldots, G_m$ covers x in φ with free variable scope β_F.*

Proof. Consider any world $\omega \in \Omega_M$ and any assignment α that is valid in ω, such that φ is well-formed in $(\text{domain}(\alpha), \beta_F)$. First, note that the scope $\text{domain}(\alpha)$ must assign types to all the free variables in φ that are not in F, such that those terms in φ that do not contain any variables in F are well-formed in $\text{domain}(\alpha)$. These are the only terms that CONSTRUCT-OGGS uses in term nodes and bounded number nodes, so we know $G_1, \ldots, G_m$ are well-formed in $\text{domain}(\alpha)$.

Now consider any assignment α_F to β_F that is valid in ω, and suppose $\omega \models_{(\alpha;\alpha_F)} \varphi$. We must show that $\alpha_F(x) \in \bigcup_{i=1}^{m} [G_i]^{\omega}_{\alpha}$. First, let $c_1, \ldots, c_m$ be the clauses (disjuncts) returned by GET-SHALLOW-DNF(φ). In order to satisfy φ, ω must satisfy at least one of $c_1, \ldots, c_m$ under $(\alpha; \alpha_F)$. Let c_i be the first clause that ω satisfies under this assignment. We will show that $\alpha_F(x)$ satisfies the target node in the corresponding OGG G_i.

To show this, we will prove the following property of the GET-OR-ADD-OGG-NODE function: if $\omega \models_{(\alpha;\alpha_F)} \ell$ for each $\ell \in L$ and GET-OR-ADD-OGG-NODE(G, τ', t, $\mathbf{L}$, F) returns a node u, then $[t]^{\omega}_{(\alpha;\alpha_F)}$ is either null or an object in $[u]^{\omega}_{\alpha}$.[3] This property is sufficient to prove the lemma because the target node in G_i is obtained by calling GET-OR-ADD-OGG-NODE(G_i, τ, x, $\mathbf{L}$, F) with $\mathbf{L}$ being the set of literals in c_i that contain x, and ω must satisfy all these literals under $(\alpha; \alpha_F)$ in order to satisfy c_i.

The proof of this property goes by induction on the depth of nesting in the tuple representation of $[t]^{\omega}_{(\alpha;\alpha_F)}$. This is really an induction on terms: the result is proven first for terms t such that $[t]^{\omega}_{(\alpha;\alpha_F)}$ is either null or a "depth-zero" object (a guaranteed object, or a non-guaranteed object with no generating objects), then for terms that denote depth-one objects, and so on. The interesting thing is that the order of induction depends on the world ω and the assignment $(\alpha; \alpha_F)$, not just some

[3] The denotation of u depends on what u's ancestors are in the graph G. But CONSTRUCT-OGGS never removes nodes from a graph, so the denotation of u cannot shrink after u is added to the graph.

syntactic properties of t. Note that when GET-OR-ADD-TYPE-NODE(G, τ, t, $\mathbf{L}$, F) calls GET-OR-ADD-OGG-NODE recursively, it passes in the term $t' = g(t)$. By the definition of possible worlds (Definition 4.10(iii)), the value of an origin function on an object o is always either null or some object in o's tuple representation, which is necessarily less deeply nested than o. Thus $[t']^{\omega}_{(\alpha;\alpha_F)}$ always has smaller depth than $[t]^{\omega}_{(\alpha;\alpha_F)}$, so (since all objects have finite depth) the induction is well-founded. The remaining details of the inductive proof are straightforward to fill in. □

4.4.5 Enumerating objects using object generation graphs

We are finally in a position to explain how the function GET-POTENTIAL-SATISFIERS-ITER works. Figure 4.15 shows a simple implementation of this function. It begins by calling CONSTRUCT-OGGS to get a list of OGGs $G_1, \ldots, G_m$; in a more sophisticated implementation, the necessary OGGs would be constructed in a preprocessing step and just retrieved when needed. The function then constructs the set of objects that satisfy at least one of $G_1, \ldots, G_m$ given the instantiation σ and the logical variable assignment α. Of course, σ may not be complete enough to determine the satisfier sets of these OGGs: the worlds $\omega \in \mathscr{F}\sigma$ may yield different values for $[G_i]^{\omega}_{\alpha}$. In this case, one of the calls to GET-OGG-NODE-SATISFIERS returns *undet*, and GET-POTENTIAL-SATISFIERS-ITER returns an iterator on which CAN-DETERMINE-NEXT immediately returns false. If σ does determine the satisfier sets of $G_1, \ldots, G_m$, then the function returns a simple iterator over the constructed set of objects.

The auxiliary function GET-OGG-NODE-SATISFIERS is shown in Figure 4.16. It includes a case for each of the five kinds of nodes that can be found in an OGG. The cases for number statement nodes and type nodes call GET-OGG-NODE-SATISFIERS recursively to get the satisfiers of their parents. Note that if the given node u is part of a cycle, then this recursion will not terminate. Also, if GET-OGG-

```
function GET-POTENTIAL-SATISFIERS-ITER(τ, x, φ, F, σ, α)
    returns an iterator over objects that may satisfy φ when bound to x
    inputs: τ, a type
            x, a logical variable
            φ, a formula
            F, a set of logical variables, including x
            σ, an instantiation of some basic random variables
            α, an assignment of values to logical variables

    (G_1, ..., G_m) ← CONSTRUCT-OGGS(τ, x, φ, F)
    S ← an empty set
    for i = 1 to m do
        u ← the target node of G_i
        satisfiers ← GET-OGG-NODE-SATISFIERS(u, σ, α)
        if satisfiers = undet
            return iterator on which first call to CAN-DETERMINE-NEXT returns false
        S ← S ∪ satisfiers
    return iterator over S
```

Figure 4.15: A simple implementation of GET-POTENTIAL-SATISFIERS-ITER.

NODE-SATISFIERS is invoked on a guaranteed object node for an infinite built-in type, then it will not terminate because it will attempt to return an infinite set.

There are three places where the behavior of GET-OGG-NODE-SATISFIERS depends on the instantiation σ and assignment α that are passed in as arguments: the call to EVAL-TERM for term nodes, the calls to EVAL-TERM for the upper-bound terms in bounded number nodes, and the calls to GET-VAR-VALUE for number variables when u is a number statement node. In each of these places, the function returns *undet* if σ does not instantiate the necessary random variables.

If all the OGGs $G_1, \ldots, G_m$ are acyclic and contain no guaranteed object nodes for infinite built-in types, then each call to GET-OGG-NODE-SATISFIERS terminates in a finite amount of time and returns a finite set (or *undet*). However, cycles and infinite guaranteed object nodes prevent this implementation of GET-POTENTIAL-SATISFIERS-ITER from terminating. In the BLOG inference engine, we use a more

```
function GET-OGG-NODE-SATISFIERS(u, σ, α)
    returns a set of objects, or undet
    inputs: u, a node in an object generation graph
            σ, an instantiation of some basic random variables
            α, an assignment of values to logical variables

    if u is a term node with term t
        val ← EVAL-TERM(t, σ, α); if val = undet return undet
        if val = null return ∅
        return {val}
    if u is a guaranteed object node for type τ
        return Guar_M(τ)
    if u is a bounded number node with upper bounds t_1, ..., t_n
        minBound ← ∞
        for i = 1 to n do
            val ← EVAL-TERM(t_i, σ, α); if val = undet return undet
            if val = null
                minBound ← 0
            minBound ← MIN(minBound, val)
        return {0, ..., minBound − 1}
    if u is a number statement node for type τ with origin functions g_1, ..., g_k
        (O_1, ..., O_k) ← a tuple of k sets, initially empty
        for i = 1 to k do
            v_i ← the source node of the edge into u labeled with g_i
            O_i ← GET-OGG-NODE-SATISFIERS(v_i, σ, α)
            if O_i = undet return undet
        S ← an empty set
        for each (o_1, ..., o_k) in O_1 × ··· × O_k do
            num ← GET-VAR-VALUE(σ, N_τ[g_1 = o_1, ..., g_k = o_k])
            if num = undet return undet
            S ← S ∪ {(τ, (g_1, o_1), ..., (g_k, o_k), n) : n ∈ {1, ..., num}}
        return S
    if u is a type node
        S ← an empty set
        for each parent v of u do
            O ← GET-OGG-NODE-SATISFIERS(v, σ, α)
            if O = undet return undet
            S ← S ∪ O
        return S
```

Figure 4.16: A function that returns the set of objects satisfying a node in an OGG.

sophisticated implementation that enumerates the objects in response to calls to CAN-DETERMINE-NEXT, rather than attempting to assemble all the objects into a set at the beginning. Thus, even if the OGGs have infinitely many satisfiers or their satisfier sets are not fully determined by σ, the iterator may still be able to return some objects that satisfy one of the OGGs in all worlds consistent with σ. If the iterator is being used to evaluate an existential formula, then EVAL-FORMULA can return true as soon as it finds some object that satisfies the formula. Thus, our more sophisticated iteration algorithm allows EVAL-FORMULA to return a true/false value in some cases where it would otherwise return *undet* or loop forever. For our purposes in this thesis, however, the simple implementation in Figures 4.15 and 4.16 is sufficient.

4.4.6 Termination conditions and correctness

We have now described all the subroutines that the top-level function EVAL-EXPR invokes as it determines the value of an expression given a partial instantiation. As we have mentioned along the way, there are cases where this computation does not terminate. To specify a condition under which it is guaranteed to terminate, we adopt the following definition.

Definition 4.19. *An object generation graph is* structurally bounded *if it satisfies the conditions of Lemma 4.9: that is, it is acyclic and it does not contain guaranteed object nodes for any infinite built-in types. This notion is extended to quantified formulas and set expressions as follows:*

- *An existentially quantified formula $\exists\ \tau\ x\ \varphi$ is structurally bounded if the graphs returned by* CONSTRUCT-OGGS*(τ, x, φ, $\{x\}$) are all structurally bounded.*
- *A universally quantified formula $\forall\ \tau\ x\ \varphi$ is structurally bounded if $\exists \tau\ x\ (\neg\varphi)$ is structurally bounded.*

- *An explicit set expression is always structurally bounded.*

- *An implicit set expression* $\{\tau$ `x :` $\varphi\}$ *is structurally bounded if the graphs returned by* CONSTRUCT-OGGS$(\tau, x, \varphi, \{x\})$ *are all structurally bounded.*

- *A tuple multiset expression* $\{t_1$`,` $\ldots$`,` t_k `for` τ_1 x_1`,` $\ldots$`,` τ_n x_n `:` $\varphi\}$ *is structurally bounded if for each* $i \in \{1, \ldots, n\}$*, the graphs returned by* CONSTRUCT-OGGS$(\tau_i, x_i, \varphi, \{x_i, \ldots, x_n\})$ *are all structurally bounded.*

- *A cardinality expression* `#`s *is structurally bounded if the implicit set expression* s *is structurally bounded.*

We can be happy to note that in the BLOG models we have used as running examples, all the quantified formulas and set expressions are structurally bounded (in fact, we have not used quantified formulas in these examples). In the urn-and-balls model (Figure 4.1), there is only one set expression, {`Ball b`}. The implicit formula in this expression is simply true, and when we call CONSTRUCT-OGGS(Ball, b, true, $\{b\}$), we get a single OGG with two nodes: a type node for Ball, and a number statement node for Ball with no origin functions. Clearly, this OGG contains no cycles or infinite guaranteed object nodes. In the hurricane model (Figure 4.2), the only set expression is {`City c`}. Here we also get a single OGG with two nodes: a type node for City, and a guaranteed object node for City. Again, this OGG is structurally bounded.

In the citation model (Figure 4.3), we find two set expressions, {`Researcher r`} and {`Publication p`}, that are analogous to the expression {`Ball b`} in the urn-and-balls-model. It is clear that they are structurally bounded. We also find the

tuple multiset expression:

```
{n, Name(NthAuthor(PubCited(c), n))
                    for NaturalNum n : n < NumAuthors(PubCited(c))}
```

This tuple multiset expression has just one variable, n, and its formula is $n <$ NumAuthors(PubCited(c)). Running Construct-OGGs on this variable and formula yields the OGG shown in Figure 4.11(b), which is structurally bounded as well.

The aircraft tracking model (Figure 4.4) does not itself contain any set expressions, but if we assert evidence using the macro discussed in Section 5.1.3, we end up with implicit set expressions such as:

```
{Blip b : (Time(b) = 8) & (b != Blip1)}
```

Construct-OGGs does nothing with the literal $b \neq$ Blip1, but it exploits the literal Time(b) $= 8$ to construct the OGG in Figure 4.10(b). This OGG is structurally bounded.

We noted above that even our simple implementation of Get-Potential-Satisfiers-Iter is guaranteed to terminate when all the OGGs it deals with satisfy the conditions of Lemma 4.9. The following lemma extends this termination result to the top-level Eval-Expr function. Note that it is not sufficient to stipulate that the expression e passed to Eval-Expr is itself structurally bounded, because a structurally bounded expression may contain a subformula that is not structurally bounded. Thus, we require that e occur in a BLOG model where every expression is structurally bounded. Also, the lemma must include an exception for unachievable instantiations, because an instantiation may agree with an achievable one on all the

variables that EVAL-EXPR accesses, but assign values to some other variables that make it unachievable.

Lemma 4.11. *Let M be a BLOG model in which all the quantified formulas and set expressions are structurally bounded. Let e be an expression in M, α be an assignment such that e is well-formed in* $\mathrm{domain}(\alpha)$*, and σ be a finite instantiation on $\mathcal{V}_M$. Then* EVAL-EXPR*(e, σ, α) terminates in a finite amount of time. If the function returns a value q other than undet, then either σ is unachievable, or σ supports e and $[e]^{\omega}_{\alpha} = q$ for all $\omega \in \mathrm{ev}\,(\sigma)$.*

Proof. We begin by showing that EVAL-EXPR(e, σ, α) terminates. First, inspection of Figures 4.7–4.9 reveals that GET-POTENTIAL-SATISFIERS-ITER is called only with certain values for its first four arguments (τ, x, φ, F). Specifically, these arguments are exactly the ones mentioned as arguments to CONSTRUCT-OBJ-GEN-GRAPH in the definition of a structurally bounded expression. So all calls to GET-POTENTIAL-SATISFIERS-ITER on M result only in structurally bounded OGGs.

Since the OGGs obtained at the beginning of GET-POTENTIAL-SATISFIERS-ITER are all structurally bounded, we know that GET-POTENTIAL-SATISIFIERS-ITER terminates in finite time. It cannot go into infinite recursion because the OGGs are acyclic, and it cannot get stuck returning an infinite set because the OGGs contain no guaranteed object nodes for infinite built-in types. Furthermore, GET-POTENTIAL-SATISFIERS-ITER returns either an iterator on which CAN-DETERMINE-NEXT returns false immediately, or an iterator over a finite set. Therefore the iteration over objects returned by this iterator in EVAL-FORMULA, EVAL-SET-EXPR, or EVAL-TUPLE-MULTISET-EXPR halts after finitely many steps. This, in turn, implies that EVAL-EXPR terminates in finite time.

Now suppose σ is achievable and EVAL-EXPR does not return *undet*. The proof that the returned value is correct basically amounts to making sure that the functions

we have defined match the definitions that they are supposed to implement. First, we can check that for any node u in a structurally bounded OGG, GET-OGG-NODE-SATISFIERS(u, σ, α) returns the satisfier set $[u]^{\omega}_{\alpha}$, if this is the same for all $\omega \in \text{ev}(\sigma)$. We can verify this by comparing the pseudocode in Figure 4.16 with Definition 4.17. Moving up a level, we know by Lemma 4.10 that the OGGs constructed at the beginning of a call to GET-POTENTIAL-SATISFIERS-ITER(τ, x, φ, F, σ, α) cover x in φ. Combining this fact with the result that GET-OGG-NODE-SATISFIERS correctly computes the satisfier set of an OGG, we can conclude that the resulting iterator includes all objects that satisfy φ when bound to x in any world consistent with σ.

Building on this result, we can show the correctness of EVAL-FORMULA, EVAL-SET-EXPR, and EVAL-TUPLE-MULTISET-EXPR, by comparison with Definitions 2.8 and 4.3. The correctness of EVAL-TERM follows by comparison with Definition 2.7. Thus EVAL-EXPR also correctly returns the value that is the denotation of e in all worlds consistent with σ. □

Thus, in BLOG models where all quantified formulas and set expressions are structurally bounded, we can be sure that all non-*undet* values returned by EVAL-EXPR are correct. However, EVAL-EXPR may still return *undet* in some cases where σ supports e. For instance, if σ instantiates V_{Slippery} to false, then it fully determines the truth value of the formula $\mathsf{Raining} \wedge \mathsf{Slippery}$. But EVAL-FORMULA would try to evaluate Raining first, and end up returning *undet* if $V_{\mathsf{Raining}} \notin \text{vars}(\sigma)$. In this sense, EVAL-EXPR is incomplete.

4.5 Structurally well-defined BLOG models

Theorem 4.7 tells us that a BLOG model is well-defined if each of its possible worlds has a supportive numbering. We showed in Section 4.3.6 that this property holds for each of the four BLOG models that we have used as examples. We also showed that in general, checking whether a given BLOG model is well-defined is undecidable (Proposition 4.8). However, it would still be useful to define a limited class of BLOG models that are known to be well-defined, along with an algorithm for checking whether a model belongs to this class. To define such a class, we will use a formalism that explicitly represents the dependencies between random variables, and the conditions under which these dependencies are active. This is the formalism of contingent Bayesian networks (CBNs), which we introduced in Section 3.5.

Recall that a CBN is a kind of PBM in which the partition for each variable is defined by a decision tree that splits on the values of certain other variables. A CBN can be represented as a directed graph in which each edge $X \rightarrow Y$ is labeled with an event, indicating the conditions under which Y's decision tree splits on X. The difficulty with CBNs, as we observed in Section 3.5.4, is that some partitions cannot be represented exactly as decision trees. In such cases, a CBN that implements a given PBM (or BLOG model) must use a decision tree that represents a strictly finer partition than the one in the PBM. There may be many decision trees that implement different refinements of the same partition. For example, if a dependency statement in a BLOG model uses the formula $\mathsf{Raining} \wedge \mathsf{Slippery}$, there is no decision tree over basic random variables that exactly implements the corresponding partition $\{\{\omega \in \Omega_M \ : \ \omega \models \mathsf{Raining} \wedge \mathsf{Slippery}\}, \{\omega \in \Omega_M \ : \ \omega \not\models \mathsf{Raining} \wedge \mathsf{Slippery}\}\}$. The

decision tree must either split on V_{Raining} first, yielding the partition:

$$\left\{ \begin{array}{l} \mathrm{ev}\left(V_{\mathsf{Raining}} = \mathsf{true}, V_{\mathsf{Slippery}} = \mathsf{true}\right), \\ \mathrm{ev}\left(V_{\mathsf{Raining}} = \mathsf{true}, V_{\mathsf{Slippery}} = \mathsf{false}\right), \\ \mathrm{ev}\left(V_{\mathsf{Raining}} = \mathsf{false}\right) \end{array} \right\}$$

or split on V_{Slippery} first, yielding another three-block partition. In general, as we saw in Section 3.5.4, some choices of decision trees may yield a structurally well-defined CBN, while others may not.

This problem is similar to the problem of choosing an evaluation strategy in a logic programming language such as Prolog. We adopt a similar solution: we define a canonical way of constructing a decision tree for a basic random variable in a given BLOG model. Our strategy is to define a procedure Get-CPD-And-Args, which takes as inputs a basic random variable X and an instantiation σ. This procedure follows the semantics of dependency (and number) statements as defined in Section 4.3.4, making calls to Eval-Expr to determine the elementary CPD and CPD argument values that are applicable for X given σ (if σ does not support X, the procedure returns the special value *undet*). The decision tree for X is then defined to split on basic variables in the order they are accessed by Get-CPD-And-Args. Specifically, on each path in the decision tree, the split order is determined by the behavior of Get-CPD-And-Args on instantiations σ consistent with that path. A path terminates when Get-CPD-And-Args returns a value; thus, each leaf in the decision tree corresponds to a particular elementary CPD and tuple of argument values, and hence to a block in the PBM partition for X.

A consequence of this approach is that the order in which the decision trees split on basic random variables is determined by the order of conjuncts and disjuncts in BLOG formulas. Thus, while the formulas $\mathsf{Raining} \wedge \mathsf{Slippery}$ and $\mathsf{Slippery} \wedge \mathsf{Raining}$

have the same semantics (both in pure logic and in the dependency statement semantics given in Definition 4.13), they yield different decision trees. Each of these decision trees is "correct", in that the partition it defines is a refinement of the partition defined by the dependency statement in the BLOG model.

4.5.1 Determining applicable CPDs and argument values

Figure 4.17 shows a function GET-CPD-AND-ARGS that takes as input a BLOG model M, a basic random variable X, and an instantiation σ, and returns either the applicable elementary CPD and argument values for X given σ, or a special value *undet* to indicate that σ does not support X. Like EVAL-EXPR, this function may fail to terminate, and may return *undet* in cases where σ actually does support X. However, if GET-CPD-AND-ARGS does terminate and return an elementary CPD and argument values, then the CPD and arguments are guaranteed to be correct. We have motivated GET-CPD-AND-ARGS just as a means for defining the canonical CBN for a BLOG model, but of course it will be useful for inference algorithms as well.

The function GET-CPD-AND-ARGS in Figure 4.17 first looks at the objects $o_1, \ldots, o_k$ that serve as indices for X: these are function arguments if X is a function application variable, and generating objects if X is a number variable. The function checks whether σ implies the existence of each non-guaranteed object o_i, by looking at the value that σ assigns to the number variable governing o_i's existence. If σ does not instantiate the governing number variable, the function returns *undet*; if σ instantiates the number variable to a value smaller than o_i's index, then it returns a tuple indicating that X takes on its "invalidity value" (null for function application variables, zero for number variables) with probability one. Note that the code does not look at other number variables — for instance, those that generate ancestors of

```
function GET-CPD-AND-ARGS(M, X, σ)
  returns an elementary CPD and argument values, or undet
  inputs: M, a BLOG model
          X, a basic random variable in M
          σ, a finite instantiation of basic random variables

  (o_1, ..., o_k) ← GET-BASIC-VAR-OBJS(X)
  for i = 1 to k do
    if IS-NON-GUAR-OBJ(o_i)
      N_o ← GET-GOVERNING-VAR(o_i)
      if N_o ∉ vars(σ)
        return undet
      if GET-VAR-VALUE(σ, N_o) < GET-OBJ-INDEX(o_i)
        return (EqualsCPD, GET-INVALIDITY-VAL(X))

  ((x_1, τ_1), ..., (x_k, τ_k)) ← GET-DEP-STMT-SCOPE(M, X)
  α ← assignment that maps (x_i, τ_i) to o_i for i = 1 to k
  (clause_1, ..., clause_n) ← GET-DEP-STMT-CLAUSES(M, X)
  for i = 1 to n − 1 do
    val ← EVAL-FORMULA(GET-COND(clause_i), σ, α)
    if val = undet return undet
    if val = true
      return GET-CPD-AND-ARGS-FROM-CLAUSE(clause_i, σ, α)
  return GET-CPD-AND-ARGS-FROM-CLAUSE(clause_n, σ, α)

---

function GET-CPD-AND-ARGS-FROM-CLAUSE(clause, σ, α)
  returns an elementary CPD and argument values, or undet
  inputs: clause, a clause from a dependency or number statement
          σ, a finite instantiation of basic random variables
          α, an assignment to the free variables in clause

  (e_1, ..., e_m) ← GET-ARG-EXPRESSIONS(clause)
  (q_1, ..., q_m) ← a new array of length m
  for j = 1 to m do
    q_j ← EVAL-EXPR(e_j, σ, α)
    if q_j = undet return undet
  return (GET-ELEM-CPD(clause), (q_1, ..., q_m))
```

Figure 4.17: Functions for finding the applicable elementary CPD and argument values for a basic random variable.

o_i in the generative process — whose values might imply that o_i does not exist. This is the one way in which GET-CPD-AND-ARGS is incomplete.

If σ implies that all of $(o_1, \ldots, o_k)$ exist, then GET-CPD-AND-ARGS gets the n clauses from X's dependency or number statement. It uses the function EVAL-FORMULA to determine if the condition for one of the first $n-1$ clauses is satisfied (recall that the condition for the last clause is always simply true). If EVAL-FORMULA returns *undet* on one of these conditions, then GET-CPD-AND-ARGS returns *undet* as well. Once the active clause is identified, the auxiliary function GET-CPD-AND-ARGS-FROM-CLAUSE is called to evaluate the argument expressions.

We can prove a termination and correctness theorem for GET-CPD-AND-ARGS, similar to the lemma we proved in the previous section for EVAL-EXPR.

Theorem 4.12. *Let M be a BLOG model in which all the quantified formulas and set expressions are structurally bounded. Let X be any basic random variable in $\mathcal{V}_M$ and σ be any finite instantiation on $\mathcal{V}_M$. Then* GET-CPD-AND-ARGS*(M, X, σ) terminates in a finite amount of time. If* GET-CPD-AND-ARGS*(M, X, σ) returns an elementary CPD c and a tuple of argument values $(q_1, \ldots, q_n)$ (rather than returning undet), then either σ is unachievable, or σ supports X in M and*

$$c(o, (q_1, \ldots, q_n)) = c_M^X(o, \lambda_M^X(\sigma))$$

for each $o \in \text{range}(X)$.

Proof. The fact that GET-CPD-AND-ARGS terminates is a direct consequence of the fact that each call to EVAL-EXPR and EVAL-FORMULA terminates, which we know from Lemma 4.11. Now suppose σ is achievable and GET-CPD-AND-ARGS does not return *undet*. The fact that the returned CPD and argument values define a distribution that matches c_M^X follows by comparing GET-CPD-AND-ARGS with

Definition 4.13, which specifies how Λ_M^X and c_M are derived from X's dependency or number statement. □

4.5.2 The canonical CBN for a BLOG model

The fact that GET-CPD-AND-ARGS accesses basic random variables in a particular order is a useful thing for us, since it allows us to define a canonical contingent Bayesian network (CBN) for a BLOG model. As we mentioned at the beginning of Section 4.5, the decision tree for each basic variable in the canonical CBN is based on the order in which GET-CPD-AND-ARGS accesses variables.

Recall that the nodes in a decision tree are finite instantiations of random variables. Each node σ *splits on* a variable $W \notin \text{vars}(\sigma)$, such that the children of σ are $\{(\sigma; W = w) : w \in \text{range}(W|\sigma)\}$ (see Definition 3.13). Decision trees for the variables in the urn-and-balls and hurricane models are shown in Figures 3.8 and 3.9. A decision tree T defines a partition Λ_T of the possible worlds, with one block for each non-truncated path starting at the root (Definition 3.14). If all the paths in T end at leaves rather than continuing infinitely, then Λ_T just consists of a block $\text{ev}(\sigma)$ for each leaf σ.

The canonical decision tree for a basic random variable X in a BLOG model M is defined by the following construction process. We start with the empty instantiation $\top$ as the root node. Then, for each node σ, we see whether GET-CPD-AND-ARGS(M, X, σ) ever calls GET-VAR-VALUE on a variable that is not in $\text{vars}(\sigma)$. If so, we let this uninstantiated variable serve as the split variable for σ, and add children to σ appropriately. Otherwise, we let σ be a leaf node. This construction process yields an infinite tree whenever some of the split variables have infinite ranges, so we cannot actually execute it in most BLOG models. Still, it provides a recursive definition of the tree. Note that this definition does not assume the calls

to GET-CPD-AND-ARGS all terminate: if GET-CPD-AND-ARGS(M, X, σ) runs forever without calling GET-VAR-VALUE on an uninstantiated variable, then σ is a leaf node.

Let us examine how this process works for the variable $V_{\mathsf{ObsColor}}[\mathsf{Draw1}]$ in the urn-and-balls model. The decision tree constructed turns out to be the same as the one on the right hand side of Figure 3.8 (although that figure uses different names for the random variables). We start by calling GET-CPD-AND-ARGS(M, $V_{\mathsf{ObsColor}}[\mathsf{Draw1}]$, $\top$). Then GET-CPD-AND-ARGS looks at the dependency statement for ObsColor, which we repeat here (from Figure 4.1) for convenience:

```
ObsColor(d)
    if (BallDrawn(d) != null) then
        ~ TabularCPD[[0.8, 0.2], [0.2, 0.8]]
                                    (TrueColor(BallDrawn(d)));
```

To determine whether the first clause is applicable, GET-CPD-AND-ARGS calls EVAL-FORMULA on $\mathsf{BallDrawn}(d) \neq \mathsf{null}$, with d bound to Draw1. This results in a call to GET-VAR-VALUE on $V_{\mathsf{BallDrawn}}[\mathsf{Draw1}]$. Since this variable is not included in the empty instantiation, it is chosen as the split variable for the root node.

Now consider the children of the root node. One child is $(V_{\mathsf{BallDrawn}}[\mathsf{Draw1}] = \mathsf{null})$. Given this instantiation, the call to EVAL-FORMULA on $\mathsf{BallDrawn}(d) \neq \mathsf{null}$ returns false, so GET-CPD-AND-ARGS-FROM-CLAUSE is called on the implicit final clause "`else = null`". This yields no more calls to GET-VAR-VALUE, so the node $(V_{\mathsf{BallDrawn}}[\mathsf{Draw1}] = \mathsf{null})$ becomes a leaf node. Each other child of the root has the form $(V_{\mathsf{BallDrawn}}[\mathsf{Draw1}] = (\mathsf{Ball}, i))$ for some natural number i. When invoked on such an instantiation, GET-CPD-AND-ARGS tries to evaluate the CPD argument $\mathsf{TrueColor}(\mathsf{BallDrawn}(d))$ (with d bound to Draw1). This results in calls to GET-VAR-VALUE on $V_{\mathsf{BallDrawn}}[\mathsf{Draw1}]$, which is already instantiated, and then on

$V_{\mathsf{TrueColor}}[(\mathsf{Ball}, i)]$, which is not instantiated and thus becomes the split variable. Note that on each path from the root, we end up splitting on the true color of a different ball. The grandchildren of the root node have the form ($V_{\mathsf{BallDrawn}}[\mathsf{Draw1}] = (\mathsf{Ball}, i)$, $V_{\mathsf{TrueColor}}[(\mathsf{Ball}, i)] = c$) for some value $c \in \{\mathsf{Blue}, \mathsf{Green}, \mathsf{null}\}$. GET-CPD-AND-ARGS successfully determines the CPD and argument values for $V_{\mathsf{ObsColor}}[\mathsf{Draw1}]$ given these instantiations, so they are leaves in the tree.

The next step in defining a CBN is to specify the CPDs. For each basic variable X, the CBN includes a CPD for X given the partition defined by X's decision tree. We will show below that if the BLOG model M contains only structurally bounded formulas and set expressions, then the canonical decision trees contain no infinite paths. Thus, the partition for X just consists of events corresponding to the leaves in X's tree. Also, under the same assumptions about M, we will show that all the leaves are instantiations on which GET-CPD-AND-ARGS returns an elementary CPD and argument values for X. Thus, we can use these return values to define a conditional distribution for X. In fact, as we will show below, this yields a CBN that correctly implements the PBM defined by the BLOG model.

We last thing to specify is the graph structure of the canonical CBN: a directed graph where edges are labeled with events indicating the conditions under which they are active. We can construct such a graph by including an edge $W \rightarrow X$ whenever the decision tree for X splits on W, and labeling it with the union of the instantiations that split on W in X's tree.

For instance, the CBN structure for the urn-and-balls example is shown in Figure 3.10. This figure uses short names for the random variables: N for N_{Ball}, C_i for $V_{\mathsf{TrueColor}}[(\mathsf{Ball}, i)]$, B_j for $V_{\mathsf{BallDrawn}}[\mathsf{Draw}j]$, and O_j for $V_{\mathsf{ObsColor}}[\mathsf{Draw}j]$. The graph includes unlabeled edges (active given the all-encompassing event Ω_M) from N_{Ball} to each TrueColor variable, because the first thing GET-CPD-AND-ARGS does on these variables is to check whether (Ball, i) exists. There are also unlabeled edges from

N_{Ball} to each BallDrawn variable. This is because GET-CPD-AND-ARGS on these variables ends up evaluating the set expression {`Ball b`}. This ultimately results in a call to GET-OGG-NODE-SATISFIERS on a number statement node for type Ball, which gets the value of N_{Ball}. For the ObsColor nodes, the incoming edges are derived from the decision tree that we discussed above: each has an unlabeled edge from the corresponding BallDrawn node, and an edge from each node $V_{\mathsf{TrueColor}}[(\mathsf{Ball}, i)]$ labeled with the event $\mathrm{ev}\left(V_{\mathsf{BallDrawn}}[\mathsf{Draw}j] = (\mathsf{Ball}, i)\right)$. Note that in general, this definition of the canonical CBN structure does not yield compact representations of the edge labels: the labels can be unions over infinite sets of instantiations.

We now give a formal definition of the canonical CBN for a BLOG model. This definition applies only to BLOG models in which all quantified formulas and set expressions are structurally bounded. On other models, the calls to GET-CPD-AND-ARGS that define our canonical decision trees may fail to terminate; and even if each call terminates or hits an uninstantiated variable, the resulting decision tree may contain infinite paths that do not identify a CPD.

Definition 4.20. *Let M be a BLOG model in which all quantified formulas and set expressions are structurally bounded. The* canonical CBN *for M, denoted $\mathcal{B}_M$, is a CBN over $\mathcal{V}_M$ defined as follows.*

i. For each basic variable X, the decision tree $T^X_{\mathcal{B}_M}$ is defined by the following construction. The root node is $\top$. For each node σ in $T^X_{\mathcal{B}_M}$:

- *if* GET-CPD-AND-ARGS*(M, X, σ) calls* GET-VAR-VALUE *on a basic variable W that is not in* $\mathrm{vars}(\sigma)$*, then the first such W serves as σ's split variable, and σ has a child $(\sigma; W = w)$ for each $w \in \mathrm{range}(W|\sigma)$;*
- *otherwise, σ is a leaf node.*

ii. For each basic variable X, the CPD $c^X_{\mathcal{B}_M}$ is defined as follows for each value

$o \in \text{range}(X)$ *and each leaf node* σ *in* $T^X_{\mathcal{B}_M}$*:*

$$c^X_{\mathcal{B}_M}(o, \text{ev}(\sigma)) = c(o, (q_1, \ldots, q_n))$$

where c *and* $(q_1, \ldots, q_n)$ *are the elementary CPD and argument values returned by* Get-CPD-And-Args*(*M*,* X*,* σ*).*

iii. The CBN structure $\mathcal{G}_{\mathcal{B}_M}$ *contains a node for each element of* $\mathcal{V}_M$*. For each pair of basic random variables* W, X *such that* W *serves as the split variable for some node in* $T^X_{\mathcal{B}_M}$*,* $\mathcal{G}_{\mathcal{B}_M}$ *contains an edge* $W \to X$ *labeled with the union of the events* $\text{ev}(\sigma)$ *for those nodes* σ *in* $T^X_{\mathcal{B}_M}$ *that have* W *as their split variable.*

The definition of the CPDs $c^X_{\mathcal{B}_M}$ assumes that the blocks in the partition $\Lambda^X_{\mathcal{B}_M}$ all correspond to leaf nodes of $T^X_{\mathcal{B}_M}$, rather than infinite paths. It also assumes that on each leaf node σ, Get-CPD-And-Args(M, X, σ) returns a CPD and tuple of arguments, rather than returning *undet* or running forever. The following lemma states that these assumptions are correct.

Lemma 4.13. *Let* M *be a discrete BLOG model in which all quantified formulas and set expressions are structurally bounded. Then the decision trees constructed by the process in Definition 4.20 contain no infinite paths. Furthermore, for each basic variable* X *and each leaf node* σ *in* $T^X_{\mathcal{B}_M}$*,* Get-CPD-And-Args*(*M*,* X*,* σ*) returns a CPD and a tuple of argument values.*

Proof. The second part of the lemma is easier, so we prove it first. Because all quantified formulas and set expressions in M are structurally bounded, we can apply Theorem 4.12 to conclude that Get-CPD-And-Args(M, X, σ) terminates in finite time for any leaf node σ. And Get-CPD-And-Args returns *undet* only when it has called Get-Var-Value on a variable W that is not instantiated in σ. If that occurred, then our construction would force σ to split on that variable W, and thus

not be a leaf node. So GET-CPD-AND-ARGS(M, X, σ) must return a CPD and a tuple of argument values.

Now assume for contradiction that in the constructed decision tree for a variable X, there is an infinite path $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \cdots$. Let σ^* be any complete instantiation that is consistent with σ_i for all $i \in \mathbb{N}$. Such an instantiation σ^* exists because the instantiations σ_i are not contradictory (we are not saying anything about whether σ^* is achievable).

In fact, the proof of this fact does not depend on the assumption that σ is finite. So if we ran GET-CPD-AND-ARGS(M, X, σ^*) with some representation for the infinite instantiation σ^*, the function would terminate after some finite time.

GET-CPD-AND-ARGS also has the property that if ρ is a sub-instantiation of σ, then the behavior of GET-CPD-AND-ARGS(M, X, ρ) is identical to that of GET-CPD-AND-ARGS(M, X, σ) up until the point (if any) where both runs of the function try to access a basic variable W that is instantiated in σ but not in ρ. Therefore, GET-CPD-AND-ARGS(M, X, σ^*) must access every basic variable that is accessed by GET-CPD-AND-ARGS(M, X, σ_i) for any σ_i on the path. By our construction of the decision tree, this implies that GET-CPD-AND-ARGS(M, X, σ^*) accesses all the basic variables that are instantiated on the path. But since the path is infinite, there are infinitely many such variables, contradicting the fact that GET-CPD-AND-ARGS(M, X, σ) terminates in finite time. □

We will now show that $\mathcal{B}_M$ *implements* the BLOG model M in the sense of Definition 3.18: that is, for each basic variable X, each block λ in $\Lambda^X_{\mathcal{B}_M}$ is a subset of some block λ' in Λ^X_M, and $c^X_{\mathcal{B}_M}(o, \lambda) = c^X_M(o, \lambda')$ for each $o \in \text{range}(X)$.

Theorem 4.14. *Let M be a discrete BLOG model in which all quantified formulas and set expressions are structurally bounded. Then the construction process in Definition 4.20 defines a CBN $\mathcal{B}_M$ that implements M.*

Proof. Lemma 4.13 ensures that the construction process in Definition 4.20 is well-defined. It is easy to check that the result is a CBN, that is, the decision trees respect the graph structure (in the sense of Definition 3.16).

Now consider any basic variable X and any block $\lambda \in \Lambda^X_{\mathcal{B}_M}$. By the definition of a CBN, $\Lambda^X_{\mathcal{B}_M}$ contains a block for each non-truncated path in $T^X_{\mathcal{B}_M}$; and by Lemma 4.13, all these non-truncated paths end at leaves. So each block $\lambda \in \Lambda^X_{\mathcal{B}_M}$ can be expressed as $\mathrm{ev}(\sigma)$ for some leaf node σ in $T^X_{\mathcal{B}_M}$. Lemma 4.13 also tells us that on this leaf node σ, GET-CPD-AND-ARGS(M, X, σ) returns a CPD and argument values. By Theorem 4.12, this implies that σ supports X in M, so $\mathrm{ev}(\sigma)$ is a subset of the block $\lambda^X_M(\sigma)$ in Λ^X_M.

The last part of Theorem 4.12 says:

$$c(o, (q_1, \ldots, q_n)) = c^X_M(o, \lambda^X_M(\sigma))$$

And the construction in Definition 4.20 specifies that:

$$c^X_{\mathcal{B}_M}(o, \mathrm{ev}(\sigma)) = c(o, (q_1, \ldots, q_n))$$

So we have $c^X_{\mathcal{B}_M}(o, \mathrm{ev}(\sigma)) = c^X_M(o, \lambda^X_M(\sigma))$, as desired. □

4.5.3 Symbol graphs

Theorem 3.17 gives criteria for determining whether a CBN is structurally well-defined, and Proposition 3.19 tells us that if a structurally well-defined CBN implements a PBM, then that PBM is well-defined as well. However, the canonical CBN for a BLOG model is typically infinite. Thus, we cannot check the criteria of Theorem 3.17 directly. Instead, we will consider a more abstract graph over function symbols and types symbols, called the *symbol graph* for the BLOG model. The sym-

bol graph is similar to the *dependency graph* used in probabilistic relational models [Koller and Pfeffer, 1998; Friedman *et al.*, 1999].

Let us say that the *corresponding symbol* for a function application variable $V_f[o_1, \ldots, o_k]$ is the function symbol f, and the corresponding symbol for a number variable $N_\tau[o_1, \ldots, o_k]$ is the type symbol τ. In the symbol graph, function symbols stand in for all the variables that correspond to them.

Definition 4.21. *The* symbol graph *for a BLOG model M is a directed graph whose nodes are the user-defined types symobls and random function symbols of M, where the parents of a type τ or function symbol f are:*

i. the random function symbols that occur on the right hand side of the dependency statement for f or some number statement for τ;

ii. the types of variables that are quantified over in formulas or set expressions on the right hand side of such a statement;

iii. the types of the arguments for f, or the return types of origin functions for τ.

The symbol graphs for our four running examples are shown in Figure 4.18. The main property that symbol graphs satisfy is the following:

Lemma 4.15. *Let $\mathcal{B}_M$ be the canonical CBN for a BLOG model M. If there is an edge (with any label) from a variable X to a variable Y in $\mathcal{G}_{\mathcal{B}_M}$, then X's corresponding symbol is an ancestor of Y's corresponding symbol in the symbol graph for M.*

Proof. The construction process for $\mathcal{G}_{\mathcal{B}_M}$ in Definition 4.20(iii) includes X as a parent of Y only if X serves as the split variable for some node in the decision tree for Y. According to part (i) of that definition, this occurs only if there is some instantiation

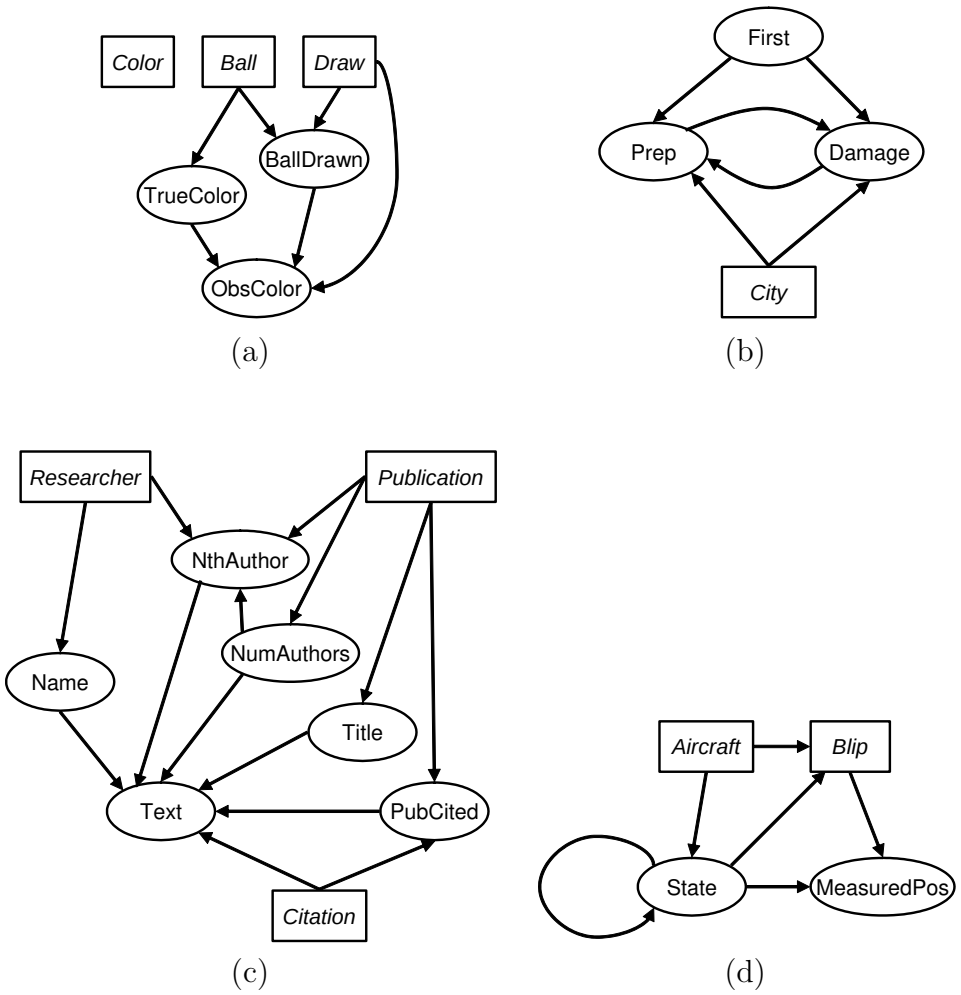


Figure 4.18: Symbol graphs for (a) the urn-and-balls model in Figure 4.1; (b) the hurricane model in Figure 4.2; (c) the bibliographic model in Figure 4.3; (d) the aircraft tracking model in Figure 4.4. We use rectangles for type nodes and ovals for function nodes.

σ such that a call to GET-CPD-AND-ARGS(M, X, σ) calls GET-VAR-VALUE on X.

Now let us consider the places in GET-CPD-AND-ARGS and its subroutines where GET-VAR-VALUE is called. One place is in the beginning of GET-CPD-AND-ARGS (Figure 4.17), where GET-VAR-VALUE is called on the number variables N_o

that govern non-guaranteed arguments o of X. Part (iii) of the definition of a symbol graph ensures that there is an edge from N_o's type symbol to the corresponding symbol for X in this case.

Another place where GET-VAR-VALUE is called is in GET-FUNC-VALUE (in Figure 4.6). It is easy to check that the only way a call to GET-CPD-AND-ARGS on X can result in a call to GET-FUNC-VALUE on f is if f appears in a term on the right hand side of X's dependency statement. In this case, part (i) of Definition 4.21 ensures that that f is a parent of X's corresponding symbol.

The last place where GET-VAR-VALUE is called is in GET-NODE-SATISFIERS, shown in Figure 4.16. Here GET-VAR-VALUE is called on a number variable to determine the satisfiers of a number statement node in an object generation graph. Suppose that during a call to GET-CPD-AND-ARGS(M, X, σ), an invocation of GET-NODE-SATISFIERS on an OGG G ends up calling GET-VAR-VALUE(σ, $N_\tau\,[o_1, \ldots, o_k]$). Let τ_0 be the type of the target node in G. Then we know that some expression on the right hand side of X's dependency statement quantifies over a logical variable of type τ_0. So by part (ii) of the definition of a symbol graph, τ_0 is a parent of X's corresponding symbol. Furthermore, inspection of the functions the construct OGGs in Figures 4.13 and 4.14 reveals that if a number statement node for type τ appears in an OGG with target type τ_0, then τ must have been reached from τ_0 by following origin functions. In other words, there must be a sequence of types $\tau_0, \tau_1, \ldots, \tau_n$ with $\tau_n = \tau$, such that each τ_{i+1} is the return type of some origin function on τ_i. By part (iii) of the symbol graph definition, this implies that there are edges $\tau_{i+1} \rightarrow \tau_i$ in the symbol graph for $i \in \{0, \ldots, n-1\}$. So $\tau_n = \tau$ is an ancestor of X's corresponding symbol. □

Given this result, it is fairly easy to prove our main theorem about symbol graphs and well-definedness.

Theorem 4.16. *Suppose M is a discrete BLOG model where:*

i. each quantified formula and set expression is structurally bounded; and

ii. the symbol graph is acyclic.

Then the canonical CBN $\mathcal{B}_M$ is structurally well-defined, and M is also well-defined. In this case, M is referred to as structurally well-defined.

Proof. To prove that $\mathcal{B}_M$ is structurally well-defined, we must show that it satisfies the conditions of Theorem 3.17. The first condition is that no consistent path in $\mathcal{G}_{\mathcal{B}_M}$ forms a cycle. In fact, we will show that $\mathcal{G}_{\mathcal{B}_M}$ contains no cycles at all. Assume for contradiction that $X_0 \leftarrow X_1 \leftarrow \cdots X_{n-1} \leftarrow X_n = X_0$ is a cycle in $\mathcal{G}_{\mathcal{B}_M}$. Then by Lemma 4.15, X_{i+1}'s corresponding symbol is an ancestor of X_i's corresponding symbol for $i \in \{0, \ldots, n-1\}$. So there is a directed path from X_0's corresponding symbol back to itself in the symbol graph, contradicting the assumption that the symbol graph is acyclic.

The second condition is that no consistent path in $\mathcal{G}_{\mathcal{B}_M}$ forms an infinite receding chain. Again, we will rule out infinite receding chains from $\mathcal{G}_{\mathcal{B}_M}$ entirely. Assume for contradiction that $X_0 \leftarrow X_1 \leftarrow X_2 \leftarrow \cdots$ is an infinite receding chain in $\mathcal{G}_{\mathcal{B}_M}$. This cannot correspond to an infinite receding chain in th symbol graph, because the number of user-defined type symbols and random function symbols in M is finite. This implies that some symbol f serves as the corresponding symbol for more than one variable on the infinite receding chain. Let us choose two such variables X_i, X_j such that $i < j$. Then there is a directed path from X_j to X_i in $\mathcal{G}_{\mathcal{B}_M}$, and so by Lemma 4.15, there is a directed path from f to f in the symbol graph. So again we contradict the assumption that the symbol graph is acyclic.

Finally, the last condition in Theorem 3.17 is that no variable has an infinite, consistent set of incoming edges in $\mathcal{G}_{\mathcal{B}}$. Assume for contradiction that X is such

a node, and let the sources of the incoming edges be $W_0, W_1, W_2, \ldots$. Since these edges are consistent, we can choose a possible world ω that activates all of them. By Definition 4.20(iii), this implies that for each $i \in \mathbb{N}$, there is an instantiation σ_i that splits on W_i in X's decision tree, such that $\omega \in \mathrm{ev}(\sigma_i)$. Now recall that in a decision tree, any two instantiations that do not lie along a common path from the root are contradictory. Since ω is consistent with all these instantiations σ_i, they must all be on the same path. Thus X's decision tree contains an infinite path. But by Lemma 4.13, this cannot occur in a model M where all quantified formulas and set expressions are structurally bounded. So we have a contradiction.

Thus, all the conditions in Theorem 3.17 are satisfied, and $\mathcal{B}_M$ is structurally well-defined. We know by Theorem 4.14 that $\mathcal{B}_M$ implements M, so by Proposition 3.19, M is well-defined as well. □

The symbol graphs for the urn-and-balls model (Figure 4.18(a)) and the citation model (Figure 4.18(c)) satisfy the acyclicity criterion, so these BLOG models are structurally well-defined. But the symbol graph for the hurricane example, shown in Figure 4.18(b), contains a cycle. So this BLOG model does not meet our criterion for being structurally well-defined. However, since this model has only finitely many basic random variables, we can check its canonical CBN (shown in Figure 3.11) and determine directly that this CBN is structurally well-defined. The problem here is that when we construct the symbol graph, we discard the edge conditions that allow us to show well-definedness for the CBN. The symbol graph for the aircraft tracking model (Figure 4.18(d)) also contains a cycle: specifically, a self-loop from State to State. The criteria in Theorem 4.16 do not exploit the fact that $\mathsf{State}(a, t)$ depends only on $\mathsf{State}(a, \mathsf{Pred}(t))$, and the nonrandom function Pred is acyclic.

Thus, our current definition of a structurally well-defined BLOG model is very conservative. When we construct the symbol graph, we ignore all the structure in

the dependency statements and just check for the occurrence of function and type symbols. In future work, we plan to extend these criteria at least to exploit acyclic functions and relations, as done in the context of probabilistic relational models by Friedman *et al.* [1999]. We may also be able to express contingent dependencies in the symbol graph, and thus handle models like the hurricane example.

When we discuss inference in the next chapter, we will state all of our results for structurally well-defined BLOG models. However, it will be clear from the proofs that all we really require is for the models to have structurally well-defined canonical CBNs. All of our running examples satisfy this less conservative criterion.

Appendix 4.A Measurability of BLOG expressions

If e is an expression (term, formula, or set expression) in a BLOG model M and α is an assignment such that e is well-formed in domain(α), then for any possible value q we can consider the event:

$$\{[e]_\alpha^\omega = q\} \triangleq \{\omega \in \Omega_M \,:\, \alpha \text{ is valid in } \omega \text{ and } [e]_\alpha^\omega = q\}$$

Note that this event excludes worlds where some of the objects in range (α) do not exist. It turns out that all such events are measurable in $\mathscr{F}_M$.

Lemma 4.17. *In a discrete BLOG model M, let e be an expression, α be an assignment such that e is well-formed in* domain(α)*, and q be a possible value of e. Then the event $\{[e]_\alpha^\omega = q\}$ is measurable in $\mathscr{F}_M$.*

Proof. For the case where e is a term, we proceed by induction on the depth of nesting in e. One base case is where e is a logical variable v. If $\alpha(v) = q$, then $\{[v]_\alpha^\omega = q\}$ is simply $\{\omega \in \Omega \,:\, \alpha \text{ is valid in } \omega\}$; otherwise it is $\varnothing$. The set of worlds where α is valid is just the intersection of the events $\{\omega \in \Omega_M \,:\, \alpha(v) \in [\tau]^\omega\}$ for

$(x, \tau) \in \text{domain}(\alpha)$, and these are basis events for $\mathscr{F}_M$. So $\{[v]_\alpha^\omega = q\}$ is measurable. The other base case is where e is a constant symbol c. Then $\{[c]_\alpha^\omega = q\}$ is the intersection of $\{\omega \in \Omega \,:\, \alpha \text{ is valid in } \omega\}$ with $\{\omega \in \Omega_M \,:\, [c]^\omega = q\}$, which is one of the basis events for $\mathscr{F}_M$. Therefore $\{[c]_\alpha^\omega = q\}$ is also measurable. Now suppose e is a term $f(t_1, \ldots, t_k)$, where the argument types of f are $a_1, \ldots, a_k$. Then for any $q \neq \mathsf{null}$:

$$\{[e]_\alpha^\omega = q\} = \bigcup_{o_1 \in \mathcal{O}_M(a_1)} \cdots \bigcup_{o_k \in \mathcal{O}_M(a_k)}$$
$$\left(\{\omega \in \Omega_M \,:\, o_1 \in [a_1]^\omega, \ldots, o_k \in [a_k]^\omega \text{ and } [f]^\omega(o_1, \ldots, o_k) = q\} \cap \bigcap_{i=1}^{k} \{[t_i]_\alpha^\omega = o_i\} \right)$$

Here the events $\{\omega \in \Omega_M \,:\, o_i \in [a_i]^\omega\}$ and $\{\omega \in \Omega_M \,:\, [f]^\omega(o_1, \ldots, o_k) = q\}$ are basis events for $\mathscr{F}_M$, and the events $\{[t_i]_\alpha^\omega = o_i\}$ are measurable by the inductive hypothesis. Also, we are assuming each set $\mathcal{O}_M(a_i)$ is countable, so the unions in the equation above are countable. Therefore $\{[e]_\alpha^\omega = q\}$ is measurable. For $q = \mathsf{null}$, we just take the union of the expression above with $\bigcup_{i=1}^k \{[t_i]_\alpha^\omega = \mathsf{null}\}$, which is also measurable.

Now consider the case where e is a formula φ. The only possible values for a formula are true and false, so for any formula φ,

$$\{[\varphi]_\alpha^\omega = \mathsf{false}\} = \{\omega \in \Omega \,:\, \alpha \text{ is valid in } \omega\} \cap \{[\varphi]_\alpha^\omega = \mathsf{true}\}^c$$

We can also swap the rules of true and false in this equation, so $\{[\varphi]_\alpha^\omega = \mathsf{true}\}$ is measurable if and only if $\{[\varphi]_\alpha^\omega = \mathsf{false}\}$ is. With this simplification in mind, we proceed by induction on the depth of the formula. The base case is where φ is an atomic formula $f(t_1, \ldots, t_k)$ where f is a Boolean function; then the result follows from our proof for terms. If φ has the form $\neg\psi$, then $\{[\varphi]_\alpha^\omega = \mathsf{true}\}$ is equal to

$\{[\psi]^\omega_\alpha = \mathsf{false}\}$, which is measurable by the inductive hypothesis. If φ has the form $\psi \wedge \chi$, then $\{[\varphi]^\omega_\alpha = \mathsf{true}\}$ is the intersection of $\{[\psi]^\omega_\alpha = \mathsf{true}\}$ and $\{[\chi]^\omega_\alpha = \mathsf{true}\}$, which are both measurable by the inductive hypothesis. Finally, suppose φ has the form $\forall\ \tau\ v\ \psi$. In this case, it is easier to show that the event $\{[\varphi]^\omega_\alpha = \mathsf{false}\}$ is measurable. The formula φ is false only in worlds where there is some binding o for v that makes ψ false.

$$\{[\varphi]^\omega_\alpha = \mathsf{false}\} = \bigcup_{o \in \mathcal{O}_M(\tau)} \{[\psi]^\omega_{(\alpha;(v,\tau)\mapsto o)} = \mathsf{false}\}$$

The events $\{[\psi]^\omega_{(\alpha;(v,\tau)\mapsto o)} = \mathsf{false}\}$ are measurable by the inductive hypothesis, and the union is countable because $\mathcal{O}_M(\tau)$ is countable. So $\{[\varphi]^\omega_\alpha = \mathsf{false}\}$ is measurable. The result follows for formulas involving $\vee$, $\rightarrow$ and $\exists$ because these can be rewritten using only $\neg$, $\wedge$ and $\forall$.

We now consider cases where e is a set expression. First suppose e is an explicit set expression $\{t_1, \ldots, t_k\}$. Then the possible values of e are multisets whose multiplicities sum to k. A particular multiset S of this kind serves as the denotation of e just in worlds where for each $o \in \mathrm{set}_S$, there are exactly $\mathrm{mult}_S(o)$ indices $i \in \{1, \ldots, k\}$ such that $[t_i]^\omega_\alpha = o$. Therefore:

$$\{[e]^\omega_\alpha = S\} = \bigcap_{o \in \mathrm{set}_S} \bigcup_{\substack{I \subseteq \{1,\ldots,k\}: \\ |I| = \mathrm{mult}_S(o)}} \left(\left(\bigcap_{i \in I} \{[t_i]^\omega_\alpha = o\} \right) \cap \left(\bigcap_{i \in \{1,\ldots,k\} \setminus I} \{[t_i]^\omega_\alpha = o\}^c \right) \right)$$

We already know that events of the form $\{[t_i]^\omega_\alpha = o\}$ are measurable, and all the unions and intersections in this equation are finite, so $\{[e]^\omega_\alpha = S\}$ is measurable when S is a multiset whose multiplicities sum to k. For all other multisets S, $\{[e]^\omega_\alpha = S\} = \varnothing$.

Now suppose e is an implicit set expression $\{\tau\ x\ :\ \varphi\}$. Then the possible values of e are finite sets, or null for cases where the denotation would otherwise be an

infinite set. A particular finite subset of $\mathcal{O}_M(\tau)$ serves as the denotation of e just in worlds where the objects in S make φ true when bound to x, and no other objects in $\mathcal{O}_M(\tau)$ do so. That is,

$$\{[e]^\omega_\alpha = S\} = \{\omega \in \Omega_M \,:\, \alpha \text{ is valid in } \omega\} \cap \left(\bigcap_{o \in S} \{[\varphi]^\omega_{(\alpha,(x,\tau)\mapsto o)} = \mathsf{true}\}\right) \cap \left(\bigcap_{o \in \mathcal{O}_M(\tau)\setminus S} \{[\varphi]^\omega_{(\alpha,(x,\tau)\mapsto o)} = \mathsf{true}\}^c\right)$$

The intersections here are countable, so $\{[e]^\omega_\alpha = S\}$ is measurable. As to $\{[e]^\omega_\alpha = \mathsf{null}\}$, it can be expressed as:

$$\{[e]^\omega_\alpha = \mathsf{null}\} = \{\omega \in \Omega_M \,:\, \alpha \text{ is valid in } \omega\} \cap \left(\bigcup_{\substack{S \subseteq \mathcal{O}_M(\tau): \\ S \text{ finite}}} \{[e]^\omega_\alpha = S\}\right)^c$$

Since the set of finite subsets of a countable set is countable, $\{[e]^\omega_\alpha = \mathsf{null}\}$ is measurable as well.

If e is a tuple multiset expression $\{t_1, \ldots, t_k \texttt{ for } \tau_1\ x_1, \ldots, \tau_n\ x_n : \varphi\}$, then its possible denotations (besides null) are finite multisets of tuples $(o_1, \ldots, o_k) \in \times_{i=1}^k (\mathcal{O}_M(r_i) \cup \{\mathsf{null}\})$, where r_i is the type of t_i. Let A be the set of all assignments $(\alpha; (x_1, \tau_1) \to o_1, \ldots, (x_n, \tau_n) \to o_n)$ such that $o_i \in \mathcal{O}_M(\tau_i)$ for $i \in \{1, \ldots, n\}$. Note that since n is finite and each set $\mathcal{O}_M(\tau_i)$ is countable, A is a countable set. A particular multiset S serves as the denotation of e just in worlds where for each object $o \in \mathrm{set}_S$, the number of assignments $\alpha' \in A$ that are valid, make φ true, and make $t_1, \ldots, t_k$ denote $o_1, \ldots, o_k$ is exactly mult_S. Let us use $\bar{o}$ to represent $(o_1, \ldots, o_k)$, and define:

$$\{\alpha' \text{ yields } \bar{o} \text{ in } e\} \triangleq \{[\varphi]^\omega_{\alpha'} = \mathsf{true}\} \cap \bigcap_{i=1}^k \{[t_i]^\omega_{\alpha'} = o_i\}$$

Given our results so far, it is clear that these sets are measurable. Now we can

express $\{[e]^\omega_\alpha = S\}$ as follows:

$$\{[e]^\omega_\alpha = S\} = \{\omega \in \Omega_M \,:\, \alpha \text{ is valid in } \omega\} \cap \bigcap_{\bar{o} \in \text{set}_S} \bigcup_{\substack{B \subseteq A\,: \\ |B| = \text{mult}_S(\bar{o})}} \left(\left(\bigcap_{\alpha' \in B} \{\alpha' \text{ yields } \bar{o} \text{ in } e\} \right) \cap \left(\bigcap_{\alpha' \in A \setminus B} \{\alpha' \text{ yields } \bar{o} \text{ in } e\}^c \right) \right)$$

The unions and intersections in this equation above are countable, so $\{[e]^\omega_\alpha = S\}$ is measurable. As in the case of implicit set expressions, we can express $\{[e]^\omega_\alpha = \mathsf{null}\}$ as the event that α is valid but $[e]^\omega_\alpha$ is not any finite subset of $\times_{i=1}^k (\mathcal{O}_M(r_i) \cup \{\mathsf{null}\}$. Since this product set is countable, it has countably many finite subsets, and $\{[e]^\omega_\alpha = \mathsf{null}\}$ is measurable.

Finally, if e is a cardinality expression $\#s$, then the possible values of e are natural numbers and null. If s is an implicit set expression for type τ, then for any natural number n,

$$\{[e]^\omega_\alpha = n\} = \bigcup_{\substack{S \subseteq \mathcal{O}_M(\tau)\,: \\ |S| = n}} \{[s]^\omega_\alpha = S\}$$

We have already shown that events of the form $\{[s]^\omega_\alpha = S\}$ are measurable, so $\{[e]^\omega_\alpha = n\}$ is measurable as well. And if $e = \#s$, then $\{[e]^\omega_\alpha = \mathsf{null}\}$ is equal to $\{[s]^\omega_\alpha = \mathsf{null}\}$. $\square$

Chapter 5

Inference for BLOG Models

In a probabilistic model, *inference* is the task of computing the posterior probability of some query event given some observed event: for example, the probability that an urn contains exactly two balls, given that we made ten draws from the urn and all the observed balls appeared blue. In Section 2.5, we mentioned several exact and approximate inference methods for probabilistic models. We focused on three sampling-based approximation algorithms: rejection sampling [Henrion, 1988], likelihood weighting [Fung and Chang, 1990; Shachter and Peot, 1990], and Markov chain Monte Carlo with a Metropolis-Hastings transition distribution [Metropolis *et al.*, 1953; Hastings, 1970].

In this chapter, we develop versions of these algorithms for approximate inference on discrete BLOG models. The main difficulty in applying the algorithms from Section 2.5 to BLOG directly is that a BLOG model may define infinitely many random variables. At the end of Section 2.5.1, we discussed Bayesian networks with infinitely many variables, observing that we can obtain correct probabilities by doing inference on a reduced BN containing just the query and evidence nodes and their ancestors. If the original BN has a topological numbering, this reduced BN is

guaranteed to be finite. We can apply the same reduction process to the canonical contingent BN $\mathcal{B}_M$ defined by a BLOG model M (see Section 4.5.2). However, the reduced CBN consisting of the query and evidence nodes and their ancestors may still be infinite. For instance, even though the urn-and-balls model in Figure 4.1 is structurally well-defined, the ObsColor nodes in its canonical CBN (Figure 3.10) have infinitely many ancestors.

As we shall see, it is possible to surmount this difficulty by exploiting the contingent dependency structure revealed by a BLOG model. Rather than having our sampling algorithms generate complete instantiations of the variables in a reduced model, we allow them to generate *partial* instantiations of the variables in the original model. For each of our three algorithms, we show that samples can be generated in finite time per sampling step, and that estimates based on the samples converge with probability one to the desired posterior probabilities as the number of samples goes to infinity. In the last section of this chapter, we describe a concrete application of our MCMC algorithm to a bibliographic citation-matching task.

5.1 Evidence and queries

Before introducing our inference algorithms, we discuss how evidence and queries can be specified in BLOG.

5.1.1 Evidence instantiations and query variables

According to the definition we used in Section 2.5, the inference task in a BLOG model M is to compute $P_M(Q|E)$, where the evidence event E and query event Q may be any events in $\mathscr{F}_M$. However, our inference algorithms will use a more structured representation of evidence and queries. We will require the evidence to

be presented as an instantiation $\mathbf{e}$ of a finite set of *evidence variables* $\mathcal{V}_E \subseteq \mathcal{V}_M$. A query event must be an instantiation $\mathbf{q}$ of a finite set of *query variables* $\mathcal{V}_Q \subseteq \mathcal{V}_M$.

In practice, we often want to compute posterior probabilities not just for a single instantiation $\mathbf{q}$, but for all instantiations of $\mathcal{V}_Q$. For instance, in the urn-and-balls example, we may wish to compute the posterior distribution for the number of balls in the urn: this involves computing the posterior probability of each instantiation of the number variable N_{Ball}. Now, this variable has infinitely many possible values (the natural numbers), so we cannot represent this posterior distribution explicitly as a table mapping each element of the range of N_{Ball} to a probability. But our algorithms are based on random sampling, and for any value of N_{Ball} that does not appear in the generated samples, they will return an approximate probability of zero. So the approximate posterior probabilities of all values of N_{Ball} an be represented in a table that explicitly assigns probabilities to the finite set of values that appeared in the samples, and implicitly assigns a probability of zero to all other values.

Thus, our algorithms will not take as input a particular instantiation of the query variables. Instead, they will just take a set of query variables $\mathcal{V}_Q$, and return a table that explicitly or implicitly assigns an approximate probability to each instantiation of $\mathcal{V}_Q$. As we will see, this takes very little extra time compared to returning a probability for a single instantiation.

5.1.2 BLOG expressions as evidence and queries

In some cases, we would like to assert observations and ask queries that do not correspond directly to basic random variables. For instance, in the urn-and-balls model, we might want to compute the probability that the sentence $\mathsf{BallDrawn}(\mathsf{Draw1}) = \mathsf{BallDrawn}(\mathsf{Draw2})$ is true. We could compute this by letting $\mathcal{V}_Q = \{V_{\mathsf{BallDrawn}}[\mathsf{Draw1}], V_{\mathsf{BallDrawn}}[\mathsf{Draw2}]\}$, and then adding up the computed probabilities of instantiations

where the two basic variables have the same value. But we would like to have an easier way of making such queries. We might also want to assert this sentence as evidence, which is impossible in the framework defined so far because there is no instantiation of the basic random variables that corresponds exactly to the event that this sentence is satisfied.

We would have no problem using BallDrawn(Draw1) = BallDrawn(Draw2) in evidence or queries if we extended the BLOG model in Figure 4.1 with an additional dependency statement:

```
random Boolean C1 = (BallDrawn(Draw1) = BallDrawn(Draw2));
```

This statement declares a random constant symbol (*i.e.*, zero-ary function symbol) C1, and asserts that it is deterministically equal to the truth value of the sentence in question. As a result, the model includes a new basic variable V_{C1} that takes the same value as the sentence in every possible world. So we can effectively condition on the sentence by using the evidence instantiation ($V_{\mathsf{C1}} = \mathsf{true}$), and query the sentence by using the query variable V_{C1}.

BLOG syntax is powerful enough to define a new basic random variable corresponding to any first-order sentence. However, we do not want to clutter up our BLOG models by including an extra random constant symbol for each sentence that we might ever use in evidence or queries. Thus, the BLOG inference engine allows new dependency statements to be added to the model automatically when a particular inference problem is presented.

Specifically, the engine allows the user to provide a file of evidence and query statements. As it reads this file, the engine adds dependency statements to the given BLOG model M to create an augmented model M', and also builds up an evidence instantiation $\mathbf{e}$ and a set of query variables $\mathcal{V}_Q$. An *evidence statement* has the form:

`obs` e `=` t`;`

where e is a term, formula, or cardinality expression[1], and t is a syntactically non-random term (and neither e nor t has any free variables). On reading this statement, the engine adds to M' the dependency statement:

`random` τ C `=` e`;`

where τ is the return type of e, and C is a new symbol that is not used elsewhere. The engine then evaluates the syntactically non-random term t, which has the same value in every world, by calling Eval-Term(t, $\top$, $\varnothing$) (see Figure 4.6). Let o be the value returned by Eval-Term. Then the engine adds the assertion $V_C = o$ to the evidence instantiation.

Thus, for example, the evidence statement:

```
obs TrueColor(BallDrawn(Draw1)) = Blue;
```

yields the dependency statement:

`random Color` C `= TrueColor(BallDrawn(Draw1));`

for some new symbol C. It also yields the evidence assertion $V_C = \mathsf{Blue}$.

We can assert the truth of a formula φ with the evidence statement:

`obs` φ `= true;`

As an abbreviation for this, we can use an alternative keyword `obstrue` and write a statement of the form:

`obstrue` φ`;`

Thus, we can assert the truth of the sentence we discussed above by writing:

[1] We cannot use a set expression here because sets are not treated as objects in BLOG; thus we cannot introduce a constant symbol that takes a set as its value.

```
obstrue BallDrawn(Draw1) = BallDrawn(Draw2);
```

A *query statement* has the form:

```
query e;
```

where e is a term, formula, or cardinality expression with no free variables. The BLOG engine interprets this statement by adding to M' the dependency statement:

```
random τ C = e;
```

where τ is the return type of e and C is a new constant symbol. The engine then adds the basic random variable V_C to the set of query variables $\mathcal{V}_Q$. Thus, for example, we can query the true color of the ball drawn on the first draw by writing:

```
query TrueColor(BallDrawn(Draw1));
```

In our implementation, the engine detects some cases where it does not actually need to add a new random constant symbol for an expression e, because there is already a basic random variable in the model that is equivalent to e. This occurs when e is a term of the form $f(t_1, \ldots, t_k)$, where f is a random function and $t_1, \ldots, t_k$ are syntactically nonrandom. Then the corresponding variable is $V_f[o_1, \ldots, o_k]$, where $o_1, \ldots, o_k$ are the values obtained by evaluating $t_1, \ldots, t_k$.

The result of adding dependency statements to a given BLOG model M is a new BLOG model M'. Adding dependency statements as described above cannot introduce any new consistent cycles into the CBN $\mathcal{B}_M$, because the added constant symbols are not used anywhere else in the model, and thus the added basic variables have no children. However, if some of the quantified formulas and cardinality expressions used in evidence or query statements are not structurally bounded (see Definition 4.19), then M' is not structurally well-defined. Thus, there are limits on the evidence and queries we can use while maintaining a structurally well-defined model.

5.1.3 Evidence about unknown objects

Although the evidence and query statements we have discussed so far are quite powerful, they are still insufficient for some cases where we need to assert evidence about objects that were not initially known to exist. In the aircraft scenario, we observe the radar blips that appear at each time step, and we need to assert their measured positions as evidence. But we cannot use an evidence statement of the form:

```
obs MeasuredPos(t) = [113, 2.9, 0.46];
```

because there are no terms t in the language of the aircraft model (Figure 4.4) that denote radar blips.

To a certain extent, existential formulas solve this problem. For instance, if we observe exactly one blip at time 8, we can write:

```
obstrue exists Blip b (Time(b) = 8
                       & MeasuredPos(b) = [113, 2.9, 0.46];
obs #{Blip b: Time(b) = 8} = 1;
```

However, this does not give us any way to refer to the observed blip later on. For instance, if we observe blips at times 7 and 8 and assert evidence about them using existential formulas, we have no straighforward way of querying whether the two blips have the same source.

We have already allowed the user to implicitly add new random constant symbols to the model to express evidence and queries. Thus, a natural solution to our current problem is to let the user introduce new symbols to refer to radar blips. These symbols will be different from the ones introduced implicitly by observation and query statements in that the user will provide names for them, and possibly use them in subsequent statements.

To introduce such symbols, we use a new kind of evidence statement called a *symbol evidence statement*. These statements are designed for the case where the user observes some new objects, introduces some new symbols, and assigns the symbols to the objects in an uninformative order. For instance, if three radar blips appear on the screen at time 8, one can assert:

```
obs {Blip b: Time(b) = 8} = {Blip1, Blip2, Blip3};
```

This statement asserts that there are exactly three radar blips at time 8, and introduces new constants Blip1, ..., Blip3 in one-to-one correspondence with those blips.

In general, a symbol evidence statement has the form:

obs $\{\tau\ x\texttt{:}\ \varphi\}$ = $\{C_1, \ldots, C_n\}$;

where τ is a type symbol, x is a logical variable, φ is a formula that is well-formed in the scope $(x \mapsto \tau)$, and $C_1, \ldots, C_n$ are distinct symbols that have not been used before. When the engine reads this statement, it adds to M' n dependency statements:

random τ C_1 $\sim$ Uniform($\{\tau\ x\ :\ \varphi\}$);
random τ C_2 $\sim$ Uniform($\{\tau\ x\ :\ \varphi$ & x != $C_1\}$);
$\vdots$
random τ C_n $\sim$ Uniform($\{\tau\ x\texttt{:}\ \varphi$ & x != C_1 & ...& x != $C_{n-1}\}$)

The engine also adds a dependency statement and asserts evidence as if it had read the evidence statement:

obs #$\{\tau\ x\texttt{:}\ \varphi\}$ = n;

Together, these statements assert that there are exactly n objects of type τ that satisfy φ when bound to x, and that these objects are assigned to the symbols

$C_1, \ldots, C_n$ by sampling without replacement. Thus, the symbols $C_1, \ldots, C_n$ denote distinct objects with probability one.

Continuing with our example, the symbol evidence statement:

```
obs {Blip b: Time(b) = 8} = {Blip1, Blip2, Blip3};
```

yields the dependency statements:

```
Blip1 ~ Uniform({Blip b : (Time(b) = 8)});
Blip2 ~ Uniform({Blip b : (Time(b) = 8) & (b != Blip1)});
Blip3 ~ Uniform({Blip b : (Time(b) = 8) & (b != Blip1) & (b != Blip2)});
```

It also has the effect of asserting:

```
obs #{Blip b : Time(b) = 8} = 3;
```

Once the model has been extended this way, the user can make assertions and queries about Blip1, Blip2, and Blip3, such as:

```
obs MeasuredPos(Blip1) = [113, 2.9, 0.46];
query Source(Blip1) = Source(Blip3);
```

The new constants introduced by a symbol evidence statement bear some resemblance to *Skolem constants*, which can be used to replace existentially quantified variables in first-order formulas (see, *e.g.,*, Chapter 9 of Russell and Norvig [2003]). However, there is an important semantic difference between existentially quantified formulas, and assertions that involve constants introduced in a symbol evidence statement. To see this, consider the simple BLOG model shown in Figure 5.1 for the price levels of wines offered in restaurants. According to this model, each restaurant offers from 6 to 15 wines. However, if a restaurant is fancy, then it offers from 1 to 5 reasonably-priced wines and from 5 to 10 expensive ones; whereas if a restaurant is not fancy, it offers from 5 to 10 reasonable wines and from 1 to 5 expensive ones.

```
1   type Restaurant;
2   guaranteed Restaurant Jacks, Zazo, Timbuktu, Lucys;
3   random Boolean IsFancy(Restaurant r) ~ Bernoulli[0.5];

4   type PriceLevel;
5   guaranteed PriceLevel Reasonable, Expensive;

6   type WineOffering;
7   origin Restaurant Provider(WineOffering);
8   origin PriceLevel Price(WineOffering);

9   #WineOffering(Provider = r, PriceLevel = p)
10      if IsFancy(r) & p = Reasonable then ~ UniformInt[1, 5]
11      elseif IsFancy(r) & p = Expensive then ~ UniformInt[5, 10]
12      elseif !IsFancy(r) & p = Reasonable then ~ UniformInt[5, 10]
13      elseif !IsFancy(r) & p = Expensive then ~ UniformInt[1, 5];
```

Figure 5.1: A pedagogical example of a BLOG model for wines offered in restaurants.

Now suppose we are looking at a wine list for Jack's restaurant and trying to figure out whether it is fancy. Consider the following evidence statements:

```
obs {WineOffering w:  Provider(w) = Jacks}
         = {W1, W2, W3, W4, W5, W6, W7};
obs PriceLevel(W1) = Expensive;
```

Recall that the denotation of W1 is chosen uniformly at random from the objects in the set {`WineOffering w:  Provider(w) = Jacks`}. Thus, asserting this evidence is equivalent to saying that we chose a wine uniformly from the wine list and observed that it was expensive. This raises the posterior probability that Jack's is fancy above 0.5. On the other hand, suppose we assert the evidence:

```
obs exists WineOffering w ((Provider(w) = Jacks)
                           & PriceLevel(w) = Expensive);
```

This statement just asserts that there is some expensive wine on the list at Jack's. The posterior given this evidence is identical to the prior, because according to the model, every restaurant has at least one expensive wine offering.

Thus, a symbol evidence statement followed by some other evidence statements that use the introduced symbols is not, in general, equivalent to an existentially quantified statement. The way we specify the evidence in a particular scenario should reflect how the evidence was obtained. In this example, if we glanced randomly at a single wine on the list and saw that it was expensive, then the first set of evidence statements is appropriate. On the other hand, if we scanned down the list checking for an expensive wine — or if a particularly high price on the menu "jumped out at us" — then the existential statement is the one to use.

5.2 Rejection sampling

Our first inference algorithm for BLOG is a version of the rejection sampling algorithm discussed in Section 2.5.1. As we noted in that section, rejection sampling is completely impractical in cases where the prior probability of the evidence is very small. The BLOG version of the algorithm has the same drawback, so it is not of much practical use. However, this algorithm illustrates how we can do inference on a BLOG model with infinitely many basic variables by sampling partial instantiations.

5.2.1 The algorithm

Pseudocode for our rejection sampling algorithm is shown in Figure 5.2. The top-level function BLOG-Rejection-Sampling, repeatedly calls Get-Sample-RS to generate the next sample σ, which is an instantiation that includes the evidence and query variables. If σ is consistent with $\mathbf{e}$, then the function increments the

```
function BLOG-Rejection-Sampling(M, e, V_Q, N)
   inputs: M, a discrete BLOG model
           e, a finite evidence instantiation on V_M
           V_Q, a set of query variables in V_M
           N, the number of samples to generate
   returns a mapping from range(V_Q) to probabilities, with implicit zeroes

   W ← a map from range(V_Q) to real numbers, with values
         lazily initialized to zero when accessed
   for j = 1 to N do
      σ ← Gen-Sample-RS(M, vars(e) ∪ V_Q)
      if σ agrees with e on vars(e)
         W[q] ← W[q] + 1 where q = σ[V_Q]
   return Normalize(W)
```

```
function Gen-Sample-RS(M, V_T)
   inputs: M, a discrete BLOG model
           V_T, a set of target variables in V_M
   returns an instantiation that includes V_T

   σ ← ⊤
   while V_T \ vars(σ) ≠ ∅ do
      σ ← Inst-One-Var(M, σ)
   return σ
```

```
function Inst-One-Var(M, σ)
   returns a version of σ with one more variable instantiated

   iter ← Get-Basic-Vars-Iter(M)
   while Has-Next(iter) do
      X ← Get-Next(iter)
      if X ∉ vars(σ)
         val ← Get-CPD-And-Args(M, X, σ)
         if val ≠ undet
            (c, (q_1, ..., q_m)) ← val
            x ← Sample(c, (q_1, ..., q_m))
            return (σ; X = x)
   error "Targets not instantiated, but no uninstantiated variable is supported."
```

Figure 5.2: Functions that implement rejection sampling for BLOG models.

count for the query variable instantiation $\sigma[\mathcal{V}_Q]$; otherwise, it rejects the sample. After generating N samples, the function returns a normalized version of the map $\mathbf{W}$. The function NORMALIZE does nothing if all the entries in the map are zero; otherwise, it divides each count in the mapping by the sum of the counts.

GET-SAMPLE-RS is a very simple function that just starts with an empty instantiation σ, and calls INST-ONE-VAR on σ repeatedly until all the target variables (*i.e.*, evidence and query variables) are instantiated. INST-ONE-VAR is more interesting: its job is to iterate over all the basic random variables in M until it finds a variable X that is not already in $\text{vars}(\sigma)$, but is supported by σ (this the same process we used to construct a supportive split tree in the proof of Theorem 3.13). Then INST-ONE-VAR assigns X a value sampled from X's CPD given σ. Note that this is a completely unguided sampling process: we simply instantiate one variable after another, trusting that eventually we will instantiate all the query and evidence variables.

INST-ONE-VAR relies on two auxiliary functions: GET-BASIC-VARS-ITER and SAMPLE. GET-BASIC-VARS-ITER returns an iterator that ranges over all the basic variables of M according to some fixed numbering of the variables. We are assuming that M is discrete, which implies that the universe of each type is countable, and hence $\mathcal{V}_M$ is countable. So there is a way to put the elements of $\mathcal{V}_M$ in one-to-one correspondence with a prefix of $\mathbb{N}$. More concretely, each basic variable $V_f[o_1, \ldots, o_k]$ or $N_\tau[g_1 = o_1, \ldots, g_k = o_k]$ can be assigned a "depth" which is the maximum of the depths of nested tuples and the magnitudes of integers among its arguments $o_1, \ldots, o_k$. The number of variables at each given depth is finite. Thus, we can enumerate first the variables at depth 0, then those at depth 1, depth 2, etc.

The function SAMPLE(c, $(q_1, \ldots, q_m)$) returns a sample from the conditional distribution $c(\cdot, (q_1, \ldots, q_m))$, such that the values returned by all calls to SAMPLE are independent. In our implementation, SAMPLE is a method that is part of the Java

interface for elementary CPDs.

5.2.2 Termination

There are several places in our rejection sampling algorithm where infinite loops could arise. One is in GEN-SAMPLE-RS, which may instantiate an infinite sequence of variables without instantiating all the variables in the target set $\mathcal{V}_T$. The subroutine INST-ONE-VAR may also go into an infinite loop, iterating over basic random variables forever without finding one that is supported by σ but not already in $\text{vars}(\sigma)$. Even the function GET-CPD-AND-ARGS may fail to terminate on some models. However, we will show that on structurally well-defined BLOG models, none of these infinite loops can occur: GEN-SAMPLE-RS returns an instantiation after a finite amount of time with probability one.

We begin by noting that for any discrete BLOG model, all the instantiations σ constructed in GEN-SAMPLE-RS have supportive numberings and are achievable. This fact will be helpful in later proofs.

Lemma 5.1. *If σ is an instantiation constructed during a run of* GEN-SAMPLE-RS*, then the order in which variables were added to σ is a supportive numbering for σ, and σ is achievable.*

Proof. At the beginning of a run of GEN-SAMPLE-RS, the instantiation σ constructed so far is the empty instantiation, which has a trivial supportive numbering and is clearly achievable. As an inductive hypothesis, suppose the value of the program variable σ after n calls to INST-ONE-VAR has the specified supportive numbering and is achievable. If INST-ONE-VAR returns an instantiation $(\sigma; X = x)$, then we know GET-CPD-AND-ARGS(M, X, σ) returned an elementary CPD c and argument values $(q_1, \ldots, q_m)$. Since we know σ is achievable, Theorem 4.12 ensures that σ supports X. Thus the new instantiation $(\sigma; X = x)$ has the desired supportive

numbering. Theorem 4.12 also tells us:

$$c(o, (q_1, \ldots, q_m)) = c_M^X(o, \lambda_M^X(\sigma))$$

for each $o \in \text{range}(X)$. Since x is sampled from $\textsc{Sample}(c, q_1, \ldots, q_m)$, it follows that $c_M^X(x, \lambda_M^X(\sigma)) > 0$. Thus $(\sigma; X = x)$ has a supportive numbering that yields no zero factors, so by Proposition 4.6, it is achievable. The inductive step is complete. □

The next lemma makes a connection between the basic variables accessed by Get-CPD-And-Args and the labeled graph structure of the canonical CBN $\mathcal{B}_M$. Recall that a variable W is an *active parent* of X given σ if there is an edge $(W \to X \mid A)$ such that σ entails A, that is, $\text{ev}(\sigma) \subseteq A$.

Lemma 5.2. *Let M be structurally well-defined BLOG model, and let $\mathcal{B}_M$ be its canonical CBN. Let X be any basic variable in $\mathcal{V}_M$ and σ be any finite, achievable instantiation on $\mathcal{V}_M$. Then all the variables on which* Get-CPD-And-Args*(M, X, σ) calls* Get-Var-Value *are active parents of X given σ in $\mathcal{B}_M$.*

Proof. Because every quantified formula and set expression in M is structurally bounded, we know by Theorem 4.12 that $\textsc{Get-CPD-And-Args}(M, X, \sigma)$ terminates in a finite amount of time. Therefore it calls Get-Var-Value on a finite number of variables. Let $W_1, W_2, \ldots, W_n$ be the distinct variables on which $\textsc{Get-CPD-And-Args}(M, X, \sigma)$ calls Get-Var-Value, in order by the first call for each variable. Also, let $\sigma_0 = \top$, and let $\sigma_i = \sigma[\{W_1, \ldots, W_i\}]$ for $i \in \{1, \ldots, n\}$. Since the behavior of GEt-CPD-And-Args depends on σ only through calls to Get-Var-Value, we know that $\textsc{Get-CPD-And-Args}(M, X, \sigma_i)$ calls Get-Var-Value on the variables $W_1, \ldots, W_{i+1}$ in that order. We will show by induction that the instantiations $\sigma_0, \ldots, \sigma_n$ are all nodes in the decision tree $T^X_{\mathcal{B}_M}$, and $W_1, \ldots, W_n$ are all active parents of X given σ in $\mathcal{B}_M$.

The base case is for σ_0 and W_1. Since $\sigma_0 = \top$, it is the root of $T^X_{\mathcal{B}_M}$. By the construction in Definition 4.20, the split variable for σ_0 is the first variable on which GET-CPD-AND-ARGS(M, X, σ_0) calls GET-VAR-VALUE, namely W_1. By the definition of a CBN, a node σ_0 in a decision tree can split on a variable W_1 only if W_1 is an active parent of X given σ_0. Therefore W_1 must be an active parent of X given σ_0, and hence given the extension σ as well.

For the inductive case, suppose σ_i is a node in $T^X_{\mathcal{B}_M}$ with W_{i+1} as its split variable, and consider $\sigma_{i+1} = (\sigma_i; W_{i+1} = \sigma_{W_{i+1}})$. Since σ is achievable, we know $\sigma_{W_{i+1}} \in \text{range}\,(W_{i+1}|\sigma_i)$, so σ_{i+1} is a child of σ_i in $T^X_{\mathcal{B}_M}$. By the same argument as in the base case, σ_{i+1} splits on W_{i+2} and W_{i+2} is an active parent of X given σ. So the induction is complete. □

We can now prove our first termination result, for the subroutine INST-ONE-VAR.

Lemma 5.3. *If M is a structurally well-defined BLOG model and σ is an achievable, incomplete instantiation of $\mathcal{V}_M$, then the function* INST-ONE-VAR*(M, σ) returns an instantiation after a finite amount of time.*

Proof. Because M is structurally well-defined, we know by Theorem 4.12 that every call to GET-CPD-AND-ARGS on M terminates in finite time. So there are two ways INST-ONE-VAR could fail to return an instantiation: it could finish iterating over all the basic variables without instantiating any of them, reaching the error statement at the end of the function; or it could iterate over the basic variables forever without instantiating any of them. In either case, it must be that GET-CPD-AND-ARGS(M, X, σ) returns *undet* on every variable $X \in \mathcal{V}_M \setminus \text{vars}\,(\sigma)$. Assume for contradiction that this is true.

Consider the context-specific graph $\mathcal{G}^\sigma_M$ derived from the CBN $\mathcal{B}_M$ by retaining only those edges that are active given σ. Because M is structurally well-defined, we

know $\mathcal{B}_M$ is structurally well-defined, and hence (by Lemma 3.18 and the fact that σ is achievable) $\mathcal{G}_M^\sigma$ has a topological numbering π_σ.

Because σ is incomplete, there is some variable in $\mathcal{V}_M \setminus \text{vars}(\sigma)$. Let Y be the first such variable according to π_σ. This implies that $\text{Pred}_{\pi_\sigma}[Y] \subseteq \text{vars}(\sigma)$, and thus because π_σ is topological, all the variables that are active parents of Y given σ are in $\text{vars}(\sigma)$. It follows by Lemma 5.2 that all the variables on which GET-CPD-ARGS(M, Y, σ) calls GET-VAR-VALUE are in $\text{vars}(\sigma)$. Thus GET-CPD-ARGS(M, Y, σ) cannot return *undet*, and we have the desired contradiction. □

We can finally prove our main termination result for this rejection sampling algorithm.

Theorem 5.4. *Let M be a structurally well-defined BLOG model, and let $\mathcal{V}_T$ be a finite subset of $\mathcal{V}_M$. Then* GEN-SAMPLE-RS*(M, $\mathcal{V}_T$) returns an instantiation in a finite amount of time.*

Proof. By Lemma 5.1, we know all the instantiations σ constructed by GEN-SAMPLE-RS(M, $\mathcal{V}_T$) are achievable and have a common supportive numbering. Because the instantiations passed into INST-ONE-VAR are all achievable (and also incomplete, because GEN-SAMPLE-RS only continues running as long as some variables in $\mathcal{V}_T$ are not instantiated), Lemma 5.3 tells us that each call to INST-ONE-VAR returns an instantiation in finite time. Thus, the only way GEN-SAMPLE-RS can run forever is if it increases the size of σ forever with calls to INST-ONE-VAR, without ever instantiating all the variables in $\mathcal{V}_T$.

Assume for contradiction that this occurs. Let $\sigma_0, \sigma_1, \sigma_2, \ldots$ be the sequence of values of the program variable σ. Now consider the infinite conjunction $\wedge_i \sigma_i$. Because the order in which the variables were instantiated is a supportive numbering for every σ_i (by Lemma 5.1), it is a supportive numbering for $\wedge_i \sigma_i$ as well; the fact that this

supportive numbering yields no zero factors also carries over to $\wedge_i \sigma_i$. So because all BLOG models respect their outcome space (Proposition 4.6), $\wedge_i \sigma_i$ is achievable.

Therefore by Lemma 3.18, the context-specific graph $\mathcal{G}_{\mathcal{B}_M}^{\wedge_i \sigma_i}$ has a topological numbering π. Because there is no i such that $\mathcal{V}_T \subseteq \text{vars}(\sigma_i)$, we know $\wedge_i \sigma_i$ is not complete. Let Y be the first variable according to π that is not in $\text{vars}(\wedge_i \sigma_i)$. Let $\text{Pa}^{\wedge_i \sigma_i}(Y)$ be the set of active parents of Y given $\wedge_i \sigma_i$. Because π is topological, we know this set is finite and is a subset of $\text{vars}(\wedge_i \sigma_i)$. Therefore there is some instantiation σ_i in the sequence such that $\text{Pa}^{\wedge_i \sigma_i}(Y) \subseteq \text{vars}(\sigma_i)$. And σ_i activates no more edges than $\wedge_i \sigma_i$ does, so it follows that $\text{Pa}^{\sigma_i}(Y) \subseteq \text{vars}(\sigma_i)$. Thus by Lemma 5.2, GET-CPD-AND-ARGS(M, Y, σ_i) will not call GET-VAR-VALUE on any variable not in $\text{vars}(\sigma_i)$; hence it will not return *undet*. The same holds for all instantiations after σ_i in the sequence.

Thus, in calls to INST-ONE-VAR after the ith one, GET-CPD-AND-ARGS(M, Y, σ) will not return *undet*. So at this point, INST-ONE-VAR will instantiate Y as soon as it reaches Y in its iteration over basic variables. The iterator returned by GET-BASIC-VARS-ITER uses some global numbering π_g of the basic variables. So after the ith call, INST-ONE-VAR can instantiate at most $\pi_g(Y)$ other variables before instantiating Y. So Y is indeed included in σ at some point, contradicting the assumption that it is not in $\text{vars}(\wedge_i \sigma_i)$. □

5.2.3 Correctness

We have now shown that on a structurally well-defined BLOG model M, each call to the function GEN-SAMPLE-RS generates a sample in a finite amount of time. But the sample is not an outcome in Ω_M: it is a finite instantiation on the basic random variables, which represents a set of outcomes. Thus, our algorithm is best viewed as a rejection sampler not on the original outcome space Ω_M, but on a space whose

elements are events in Ω_M. Specifically, these events correspond to instantiations returned by GEN-SAMPLE-RS. We will call this set of events Σ.

We begin our correctness proof by showing that the proposal distribution defined by GEN-SAMPLE-RS has certain properties.

Lemma 5.5. *Let M be a structurally well-defined BLOG model, $\mathbf{e}$ be a finite evidence instantiation, and $\mathcal{V}_Q$ be a finite set of query variables. Let q be the distribution on return values defined by* GEN-SAMPLE-RS*$(M, \text{vars}(\mathbf{e}) \cup \mathcal{V}_Q)$. Let Σ be the set of instantiations that have a positive probability of being returned. Then:*

i. For each instantiation $\sigma \in \Sigma$, $q(\sigma) = P_M(\sigma)$.

ii. The instantiations in Σ are mutually disjoint.

iii. For each instantiation $\sigma \in \Sigma$, either $\text{ev}(\sigma) \subseteq \text{ev}(\mathbf{e})$ or $\text{ev}(\sigma) \cap \text{ev}(\mathbf{e}) = \varnothing$.

iv. For each instantiation $\sigma \in \Sigma$ and each instantiation $\mathbf{q}$ of $\mathcal{V}_Q$, either $\text{ev}(\sigma) \subseteq \text{ev}(\mathbf{q})$ or $\text{ev}(\sigma) \cap \text{ev}(\mathbf{q}) = \varnothing$.

Proof. The possible runs of GEN-SAMPLE-RS can be thought of as paths through a tree, where the nodes are instantiations that serve as values for the variable σ in GEN-SAMPLE-RS. All runs start at the root node, which is the empty instantiation. At each node σ, the tree splits on the variable X found by INST-ONE-VAR; then the children of σ have the form $(\sigma; X = x)$. A particular run corresponds to a path from the root of the tree. A run terminates when the evidence and query variables have all been instantiated; we showed in Theorem 5.4 that this always occurs after finitely many steps. Because the decision about whether to return an instantiation or keep extending it is made deterministically based on whether the instantiation includes all the target nodes, the instantiations that get returned are always leaves in the tree: no trajectories go beyond them. These leaf nodes form the set Σ.

There cannot be more than one path from the root of this tree to the same instantiation, because two paths that diverge at some node assign contradictory values to the variable that was split on at that node. Thus, for any instantiation $\sigma \in \Sigma$, the proposal probability $q(\sigma)$ is the probability of the unique path that reaches σ. If a node τ in the tree splits on a variable X, then the SAMPLE function generates a particular value $o \in \text{range}(X)$ with probability $c(o, (q_1, \ldots, q_m))$, where $(c, q_1, \ldots, q_m)$ is the value returned by GET-CPD-AND-ARGS(M, X, τ). By Theorem 4.12, this probability is equal to $c_M^X(o, \lambda_M^X(\tau))$. So the probability of a leaf node σ being returned is:

$$q(\sigma) = \prod_{X \in \text{vars}(\sigma)} c_M^X(\sigma_X, \lambda_M^X(\sigma))$$

By Lemma 3.7, this is equal to $P_M(\sigma)$, so we have proven part (i).

For part (ii), consider any two distinct leaf nodes σ and σ'. Let ρ be the last node that is on the paths from the root to both σ and σ'. Then σ and σ' disagree on the value of the variable that ρ splits on, so they define disjoint events. For parts (iii) and (iv), note that an instantiation is returned only if it assigns values to all the query and evidence variables. If the values assigned to these variables by σ match $\mathbf{e}$ or $\mathbf{q}$ exactly, then $\text{ev}(\sigma) \subseteq \text{ev}(\mathbf{e})$ or $\text{ev}(\sigma) \subseteq \text{ev}(\mathbf{q})$. Otherwise, σ must disagree with the evidence or query insantiation on some variable, and thus defines a disjoint event. □

The next step is to prove a convergence result for the estimates returned by BLOG-REJECTION-SAMPLING. To apply our results from Section 2.5.1, we need to view BLOG-REJECTION-SAMPLING as a rejection sampler on Σ. This requires some additional notation. Let $\widetilde{E}$ be the set of events in Σ that are consistent with the evidence: $\widetilde{E} = \{\sigma \in \Sigma : \text{ev}(\sigma) \subseteq \text{ev}(\mathbf{e})\}$. For any given instantiation $\mathbf{q}$ of the query variables, define the query event $\widetilde{Q}$ analogously. We know from Lemma 5.5(i) that the probability measure q on Σ defined by sampling from GEN-SAMPLE-RS

has the property that $q(\sigma) = P_M(\sigma)$. The notational similarity here obscures the difference between q and P_M: P_M is a probability measure on possible worlds, while q is a probability measure on events represented by instantiations. Thus, we can write $q(\widetilde{E})$ to denote the sum of the probabilities of all the events consistent with the evidence; $P_M(\widetilde{E})$ is not well-formed. We also define $\widetilde{P}_{\widetilde{E}}$ to be the probability measure on $\widetilde{E}$ such that $\widetilde{P}_{\widetilde{E}}(\sigma) = q(\sigma)/q(\widetilde{E})$.

Lemma 5.6. *Let M be a structurally well-defined BLOG model, $\mathbf{e}$ be a finite evidence instantiation with $P_M(\mathbf{e}) > 0$, $\mathcal{V}_Q$ be a finite set of query variables, and $\mathbf{q}$ be any instantiation of $\mathcal{V}_Q$. Let $\widetilde{P}$, $\widetilde{E}$, and $\widetilde{Q}$ be derived from P_M, $\mathbf{e}$, and $\mathbf{q}$ as described above. Then the table entry for $\mathbf{q}$ returned by* BLOG-Rejection-Sampling*(M, $\mathbf{e}$, $\mathcal{V}_Q$, N) converges with probability one to $\widetilde{P}_{\widetilde{E}}(\widetilde{Q} \cap \widetilde{E})$ as $N \to \infty$.*

Proof. The instantiations returned by Gen-Sample-RS are independent samples from the proposal distribution q on Σ. By definition, $\widetilde{P}_{\widetilde{E}}(\sigma)$ is proportional to $q(\sigma)$ on $\widetilde{E}$, with proportionality constant $\frac{1}{q(\widetilde{E})}$. In BLOG-Rejection-Sampling, a sample σ is tallied for some query instantiation if it is in $\widetilde{E}$, and tallied for the particular query instantiation $\mathbf{q}$ if it is in $\widetilde{Q} \cap \widetilde{E}$. The Normalize function divides all counts by the sum of the counts at the end, which reproduces the estimate in Equation 2.6 because the sum of the counts is the number of samples that were in $\widetilde{E}$. Thus, all the conditions of Proposition 2.13 are satisfied, so the convergence property holds. □

This convergence result may seem different from the one we want to prove: our goal was to approximate $P_M(\mathbf{q}|\mathbf{e})$, not $\widetilde{P}_{\widetilde{E}}(\widetilde{Q} \cap \widetilde{E})$. It turns out, however, that these quantities are the same. To prove this, we will use the following lemma. Note the correspondence between the conditions in this lemma and the properties we proved in Lemma 5.5.

Lemma 5.7. *Let P be a probability measure on a countable set Ω, and let Σ be a*

set of non-empty subsets of Ω*. Also, let* A *be a subset of* Ω *and define* $\widetilde{A} \triangleq \{s \in \Sigma : s \subseteq A\}$*. If:*

i. $\sum_{(s \in \Sigma)} P(s) = 1$*;*

ii. the sets in Σ *are mutually disjoint; and*

iii. for each $s \in \Sigma$*, either* $s \subseteq A$ *or* $s \cap A = \varnothing$*,*

then $\sum_{(s \in \widetilde{A})} P(s) = P(A)$*.*

Proof. Because the sets in Σ are mutually disjoint, we know:

$$\sum_{s \in \widetilde{A}} P(s) = P\left(\bigcup_{s \in \widetilde{A}} s\right)$$

And since each s in $\widetilde{A}$ is a subset of A, we know $\bigcup_{(s \in \widetilde{A})} s \subseteq A$. Therefore:

$$\sum_{s \in \widetilde{A}} P(s) \leq P(A)$$

Now let us introduce the sets $B = \Omega \setminus A$ and $\widetilde{B} = \{s \in \Sigma : s \subseteq B\}$. We can repeat the argument above for $\widetilde{B}$ and B, yielding:

$$\sum_{s \in \widetilde{B}} P(s) \leq P(B)$$

But A and B form a partition of Ω by definition, and $\widetilde{A}$ and $\widetilde{B}$ form a partition of Σ by condition (iii). So we can rewrite our last inequality as:

$$\sum_{s \in \Sigma} P(s) - \sum_{s \in \widetilde{A}} P(s) \leq 1 - P(A)$$

By condition (i), the first sum on the left hand side of this inequality is 1. So a bit of algebra yields:

$$\sum_{s \in \widetilde{A}} P(s) \geq P(A)$$

Combined with the reverse result that we got earlier, this yields the desired equality. □

We are now in a position to prove our main theorem about our rejection sampling algorithm.

Theorem 5.8. *Let M be a structurally well-defined BLOG model. Then for any finite evidence instantiation $\mathbf{e}$, any finite set of query variables $\mathcal{V}_Q \subseteq \mathcal{V}_M$, and any instantiation $\mathbf{q}$ of $\mathcal{V}_Q$, the probability estimates for $\mathbf{q}$ returned by* BLOG-Rejection-Sampling*$(M, \mathbf{e}, \mathcal{V}_Q, N)$ converge to the posterior probability $P_M(\mathbf{q}|\mathbf{e})$, taking finite time per sampling step.*

Proof. The fact that every sampling step terminates in finite time is given by Theorem 5.4. We also know from Lemma 5.6 that the estimates in question converge to $\widetilde{P}_{\widetilde{E}}(\widetilde{Q} \cap \widetilde{E})$. We just need to check that this result, obtained by sampling events, is still correct at the level of individual outcomes. We can write out the value to which the estimates converge more explicitly as:

$$\begin{aligned} \widetilde{P}_{\widetilde{E}}(\widetilde{Q} \cap \widetilde{E}) &= \frac{q(\widetilde{Q} \cap \widetilde{E})}{q(\widetilde{E})} \\ &= \frac{\sum_{(\sigma \in \widetilde{Q} \cap \widetilde{E})} P_M(\sigma)}{\sum_{(\sigma \in \widetilde{E})} P_M(\sigma)} \end{aligned}$$

Here we have used the result from Lemma 5.5(i) that $q(\sigma) = P_M(\sigma)$.

We will use Lemma 5.7 to show that the sum in the numerator is equal to $P_M(\mathbf{q}; \mathbf{e})$, and the denominator is equal to $P_M(\mathbf{e})$. We must check that the conditions

for applying that lemma are satisfied. The outcome space called Σ in the lemma will be the same as the outcome space we have been calling Σ, namely the set of all instantiations that can be returned by GEN-SAMPLE-RS. We know the proposal distribution q is a probability distribution on Σ, so the first condition in Lemma 5.7 is satisfied. The second condition in Lemma 5.7 is guaranteed by the second property in Lemma 5.5. The last condition in Lemma 5.7 depends on the particular set A that we use. For the numerator we will use $\text{ev}(\mathbf{q}; \mathbf{e})$ as A, which means $\widetilde{A}$ is $\widetilde{Q} \cap \widetilde{E}$; for the denominator we will use $\text{ev}(\mathbf{e})$ as A, so that $\widetilde{A}$ is $\widetilde{E}$. In both cases, condition (iii) is guaranteed by the last two properties in Lemma 5.5.

So by two applications of Lemma 5.7, we get:

$$\widetilde{P}_{\widetilde{E}}(\widetilde{Q} \cap \widetilde{E}) \;=\; \frac{P_M(\mathbf{q}; \mathbf{e})}{P_M(\mathbf{e})}$$

as desired. □

5.3 Likelihood weighting

The rejection sampling algorithm described in the previous section suffers from two serious drawbacks. One of these drawbacks is common to rejection sampling algorithms in general: as the number of observed variables increases, the probability of the evidence decreases exponentially, and thus the fraction of the samples that are accepted becomes exponentially small. But our algorithm for rejection sampling on partial instantiations has an additional drawback: the choice of which variable to instantiate next is determined solely by the variable ordering defined by GET-BASIC-VARS-ITER, subject to the constraint that variables cannot be instantiated until they are supported. The query and evidence play no role in determining which variable to instantiate next. Thus, the algorithm often ends up instantiating far

more variables than it needs to in order to support the query and evidence variables.

To remedy both these drawbacks, we propose a guided form of likelihood weighting over partial instantiations. Recall from Section 2.5.2 that in a likelihood weighting algorithm, we sample values for the non-evidence variables according to their CPDs, and deterministically instantiate the evidence variables to their observed values. We then assign the sample a weight, which is the product of the probabilities of the evidence variables given their parents. In our case, the samples are self-supporting finite instantiations. The main innovation is that in order to avoid instantiating variables unnecessarily, we maintain a stack of variables that we know we must instantiate in order to obtain a self-supporting instantiation that includes the query and evidence variables.

Pseudocode for our algorithm is shown in Figure 5.3. The top-level function BLOG-Likelihood-Weighting simply gets weighted samples from Gen-Sample-LW and tallies the results; most of the action is in Gen-Sample-LW. This function begins with an empty instantiation and an empty stack of variables to instantiate. It then pushes one of the query or evidence variables onto the stack — it is clear that these variables will need to be instantiated at some point. Now the algorithm enters the central "**do**...**until**" loop. Each time through this loop, the function looks at the variable X on the top of the stack, and tries to determine its applicable CPD and argument values given the current instantiation σ. If σ does not support X, then Get-CPD-And-Args will end up calling Get-Var-Value on a variable that is not instantiated, and returning *undet*. Now comes the main trick: the algorithm identifies the uninstantiated variable V that was passed to Get-Var-Value, and pushes it onto the stack of variables to instantiate. This makes sense because Get-CPD-And-Args will always return *undet* on X until V is added to σ. If V is not yet supported either, then the algorithm pushes another variable onto the stack, and so on. This inner loop terminates when the algorithm finds that the variable on the

```
function BLOG-Likelihood-Weighting(M, e, V_Q, N)
  inputs: M, a discrete BLOG model
          e, a finite evidence instantiation on V_M
          V_Q, a set of query variables in V_M
          N, the number of samples to generate
  returns a mapping from range(V_Q) to probabilities, with implicit zeroes

  W ← a map from range(V_Q) to real numbers, with values
      lazily initialized to zero when accessed
  for j = 1 to N do
      (σ, w) ← Gen-Sample-LW(M, e, V_Q)
      W[q] ← W[q] + w where q = σ[V_Q]
  return Normalize(W)
```

```
function Gen-Sample-LW(M, e, V_Q)
  returns an instantiation and a weight

  σ ← ⊤; stack ← an empty stack; w ← 1
  loop
    if stack is empty
      if some X in (V_Q ∪ vars(e)) is not in vars(σ)
        Push(X, stack)
      else
        return (σ, w)

    do
      X ← Peek(stack)
      val ← Get-CPD-And-Args(M, X, σ)
      if val = undet
        V ← last uninstantiated variable on which Get-Var-Value was called
        Push(stack, V)
    until val ≠ undet

    X ← Pop(stack)
    (c, (q_1, ..., q_m)) ← val
    if X in vars(e)
      x ← e_X
      w ← w × Get-Prob(c, x, (q_1, ..., q_m))
      if w = 0 return (σ, w)
    else
      x ← Sample(c, (q_1, ..., q_m))
    σ ← (σ; X = x)
```

Figure 5.3: Likelihood weighting algorithm for BLOG.

top of the stack is supported by σ.

At this point, the algorithm pops the top variable X off the stack. Next come the lines that distinguish this as a likelihood weighting algorithm: if X is an evidence variable, then the function sets it equal to its observed value, rather than sampling a value from its CPD. To account for this biased sampling, the probability of X's observed value given its parent values in σ is multiplied into the weight w. On the other hand, if X is not an evidence variable, then the algorithm samples a value for it using its CPD as usual.

As an example, consider the balls-and-urn model in Figure 4.1. If we want to query the number variable N_{Ball} given some color observations, the algorithm begins by pushing N_{Ball} onto the stack. Since N_{Ball} (which has no parents) is supported by the empty instantiation, it is immediately removed from the stack and sampled. Next, the first evidence variable $V_{\mathsf{ObsColor}}[\mathsf{Draw1}]$ is pushed onto the stack. A call to GET-CPD-AND-ARGS on $V_{\mathsf{ObsColor}}[\mathsf{Draw1}]$ returns *undet* because $V_{\mathsf{BallDrawn}}[\mathsf{Draw1}]$ needs to be instantiated. So $V_{\mathsf{BallDrawn}}[\mathsf{Draw1}]$ is added to the stack, and is sampled immediately because it is supported by σ (which now includes N_{Ball}). Now $V_{\mathsf{ObsColor}}[\mathsf{Draw1}]$ is on the top of the stack again, and the algorithm finds that $V_{\mathsf{TrueColor}}[(\mathsf{Ball}, n)]$ (for n equal to the sampled value of $V_{\mathsf{BallDrawn}}[\mathsf{Draw1}]$) is the next variable that needs to be instantiated. This variable can be sampled immediately. Now $V_{\mathsf{ObsColor}}[\mathsf{Draw1}]$ is finally supported by σ, so it is removed from the stack and instantiated to its observed value. This process is repeated for all the observations. The resulting sample will get a high weight if the sampled true colors for the balls match the observed colors.

Intuitively, this algorithm is the same as standard likelihood weighting, in that we sample the variables in some topological order. The difference is that we sample only those variables that are needed to support the query and evidence variables, and we do not bother sampling any of the other basic variables of the BLOG model.

Since the weight for a sample only depends on the conditional probabilities of the evidence variables, sampling additional variables would have no effect.

Lemma 5.9. *Let M be a structurally well-defined BLOG model, $\mathbf{e}$ be a finite evidence instantiation, and $\mathcal{V}_Q$ be a finite set of query variables. Then each call to* Gen-Sample-LW*$(M, \mathbf{e}, \mathcal{V}_Q)$ returns a weighted sample after a finite amount of time.*

Proof. There are two loops in Gen-Sample-LW that could run forever. Let us look first at the inner loop, "**do**...**until** *val* $\neq$ undet". Note that the instantiation σ does not change inside this loop; it is also possible to show that the σ is always achievable when this part of the code is reached (the argument is similar to that for Lemma 5.1, with the additional note that if setting an evidence variable to its observed value yields an unachievable instantiation, the weight w will be zero and Gen-Sample-LW will return immediately).

Assume for contradiction that some execution of this loop does not terminate. Let $V_0, V_1, V_2, \ldots$ be the infinite sequence of variables that are pushed onto the stack. Note that V_{i+1} is pushed onto the stack only if Get-CPD-And-Args(M, V_i, σ) calls Get-Var-Value on V_{i+1}. By Lemma 5.2, V_{i+1} is an active parent of V_i in $\mathcal{B}_M$ given σ. Thus the sequence $V_0, V_1, V_2, \ldots$ forms a path $V_0 \leftarrow V_1 \leftarrow V_2 \leftarrow \cdots$ consisting of edges that are activated by σ. If some variable occurs more than once in this sequence, then it is a cycle; otherwise it is an infinite receding chain. But by Lemma 3.18, the context-specific graph gor any achievable instantiation in a structurally well-defined CBN has a topological numbering, so we have a contradiction.

Now consider the outer loop in Gen-Sample-LW, which adds a variable to σ each time it is executed. If this loop runs forever, then it generates an infinite sequence of instantiations $\sigma_0, \sigma_1, \sigma_2, \ldots$. The infinite conjunction $\wedge_i \sigma_i$ has a supportive numbering that yields no zero factors, so by Proposition 4.6, it is achievable. By an extension of our argument in the previous paragraph, it is easy to show that

every variable pushed onto the stack is an active ancestor of some query or evidence variable given $\wedge_i \sigma_i$. But this set of active ancestors cannot be infinite, because that would imply that one of the finitely many query and evidence nodes has infinitely many active ancestors, contradicting the existence of a topological numbering on the context-specific graph $\mathcal{G}_{\mathcal{B}_M}^{\wedge_i \sigma_i}$. So the loop must stop after a finite amount of time. □

Theorem 5.10. *Let M be a structurally well-defined BLOG model. Then for any finite evidence instantiation $\mathbf{e}$ with $P_M(\mathbf{e}) > 0$, any finite set of query variables $\mathcal{V}_Q \subseteq \mathcal{V}_M$, and any instantiation $\mathbf{q}$ of $\mathcal{V}_Q$, the probability estimates for $\mathbf{q}$ returned by* BLOG-Likelihood-Weighting*$(M, \mathbf{e}, \mathcal{V}_Q, N)$ converge to the posterior probability $P_M(\mathbf{q}|\mathbf{e})$, taking finite time per sampling step.*

Proof. The fact that each sampling step takes finite time is given by Lemma 5.9. To prove correctness, let Σ be the set of instantiations that have a positive probability of being returned by Gen-Sample-LW. A proof similar to that of Lemma 5.5 shows that the instantiations in Σ define disjoint sets, and the probability of an instantiation $\sigma \in \Sigma$ being returned is:

$$q(\sigma) = \prod_{X \in \text{vars}(\sigma) \setminus \mathcal{V}_E} c_M^X(\sigma_X, \lambda_M^X(\sigma))$$

where $\mathcal{V}_E = \text{vars}(\mathbf{e})$. It is also straightforward to show that the weight on an instantiation σ is:

$$w(\sigma) = \prod_{X \in \mathcal{V}_E} c_M^X(\sigma_X, \lambda_M^X(\sigma))$$

As in the rejection sampling section, we define $\widetilde{E}$ to be $\{\sigma \in \Sigma : \text{ev}(\sigma) \subseteq \text{ev}(\mathbf{e})\}$. In the likelihood weighting case, every instantiation that is returned satisfies the evidence, so $\widetilde{E} = \Sigma$. We also define $\widetilde{Q}$ in an analogous way based on $\mathbf{q}$. Now define

$\widetilde{P}_{\widetilde{E}}$ to be the probability measure on $\widetilde{E}$ such that:

$$\widetilde{P}_{\widetilde{E}}(\sigma) = \frac{P_M(\sigma)}{\sum_{(\sigma \in \widetilde{E})} P_M(\sigma)}$$

We claim that Gen-Sample-LW implements importance sampling (as described in Section 2.5.2) with $\widetilde{P}_{\widetilde{E}}$ as the target distribution. To see this, note that by Lemma 3.7,

$$P_M(\sigma) = \prod_{X \in \text{vars}(\sigma)} c_M^X(\sigma_X, \lambda_M^X(\sigma))$$

So we have:

$$\begin{aligned} w(\sigma) &= \frac{P_M(\sigma)}{q(\sigma)} \\ &= \frac{\beta \widetilde{P}_{\widetilde{E}}(\sigma)}{q(\sigma)} \end{aligned}$$

where $\beta = \left(\sum_{(\sigma \in \widetilde{E})} P_M(\sigma)\right)^{-1}$. Thus, $w(\sigma)$ satisfies Equation 2.8, and the conditions of Theorem 2.14 are satisfied. Thus we know that the estimates returned by Gen-Sample-LW converge with probability one to $\widetilde{P}_{\widetilde{E}}(\widetilde{Q} \cap \widetilde{E})$ as $N \to \infty$.

We will now apply Lemma 5.7 to bring this result out of the realm of instantiations and back to the realm of possible worlds. We can write the limit of the estimates more explicitly as:

$$\widetilde{P}_{\widetilde{E}}(\widetilde{Q} \cap \widetilde{E}) = \frac{\sum_{(\sigma \in \widetilde{Q} \cap \widetilde{E})} P_M(\sigma)}{\sum_{(\sigma \in \widetilde{E})} P_M(\sigma)}$$

Lemma 5.7 allows us to show that $\sum_{(\sigma \in \widetilde{Q} \cap \widetilde{E})} P_M(\sigma) = P_M(\mathbf{e}; \mathbf{q})$, and $\sum_{(\sigma \in \widetilde{E})} P_M(\sigma) = P_M(\mathbf{e})$. Thus we have the estimates converging to $P_M(\mathbf{e}; \mathbf{q})/P_M(\mathbf{e})$, as desired. □

5.3.1 Experiments

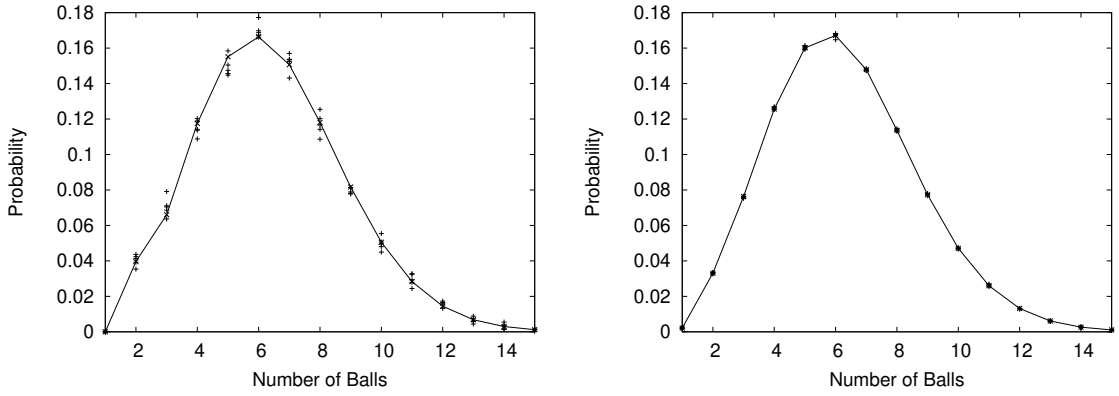
We ran two sets of experiments using the likelihood weighting algorithm of Figure 5.3. Both use the urn-and-balls model shown in Figure 4.1. The first experiment estimates the number of balls in the urn given the colors observed on 10 draws; the second experiment is an identity uncertainty problem. In both cases, we run experiments with both a noiseless sensor model, where the observed colors of balls always match their true colors, and a noisy sensor model, where with probability 0.2 the wrong color is reported.

The purpose of these experiments is to show that inference over an infinite number of variables can be done using a general algorithm in finite time. We show convergence of our results to the correct values, which were computed by enumerating equivalence classes of outcomes with up to 100 balls. More efficient sampling algorithms for these problems have been designed by hand [Pasula, 2003]; however, our algorithm is general-purpose, so it needs no modification to be applied to a different domain.

Number of balls given balanced observations. In the first experiment, we are predicting the total number of balls in the urn. The prior over the number of balls is a Poisson distribution with mean 6; each ball is blue with probability 0.5 and green otherwise. The evidence consists of color observations for 10 draws from the urn: five are blue and five are green. For each observation model, five independent trials were run, each of 5 million samples.[2]

Fig. 5.4 shows the posterior probabilities for total numbers of balls from 1 to 15 computed in each of the five trials, along with the exact probabilities. The results are all quite close to the true probability, especially in the noisy-observation case.

[2] Our Java implementation averages about 1700 samples/sec. for the exact observation case and 1100 samples/sec. for the noisy observation model on a 3.2 GHz Intel Pentium 4.



(a) Noise-free model

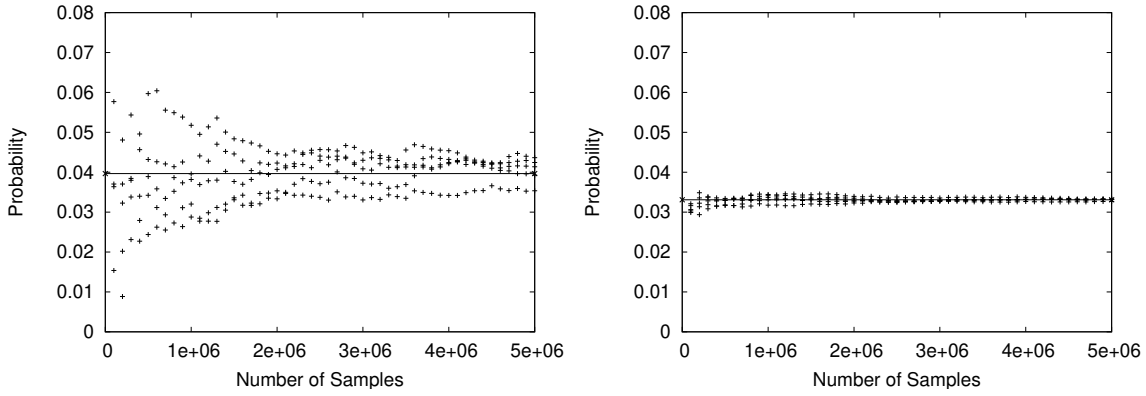
(b) Noisy model

Figure 5.4: Posterior distributions for the total number of balls given 10 observations (5 blue, 5 green) with two observation models. Exact probabilities are denoted by '×'s and connected with a line; estimates from 5 sampling runs are marked with '+'s.

The variance is higher for the noise-free model because the sampled true colors for the balls are often inconsistent with the observed colors, so many samples have zero weights.

Fig. 5.5 shows how quickly our algorithm converges to the correct value for a particular probability, $P(N = 2|\text{obs})$. The run with deterministic observations stays within 0.01 of the true probability after 2 million samples. The noisy-observation run converges faster, in just 100,000 samples.

Number of balls given all-blue observations. The posterior distributions obtained in the previous experiment are quite close to the prior, because observing five blue balls and five green ones does not tell us much about the number of balls in the urn (it does slightly favor even numbers of balls over odd numbers, and makes it very unlikely that there is just one ball). To see how the likelihood weighting algorithm performs when the posterior is more significantly different from the prior, we assert



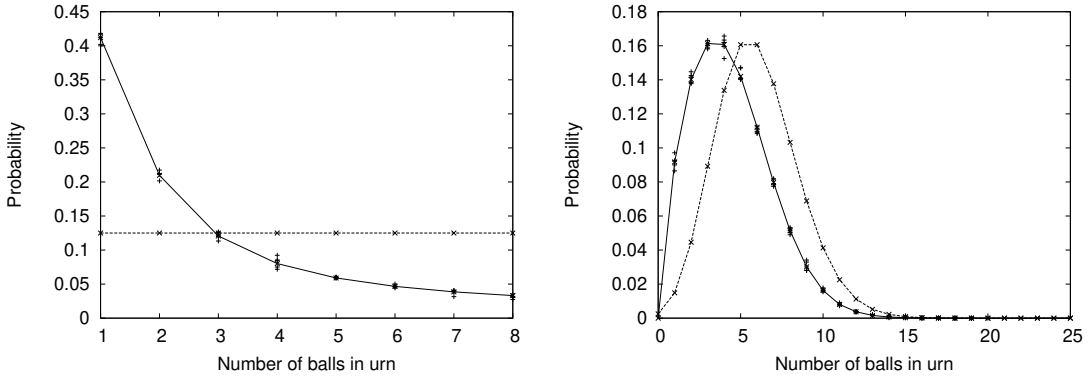
(a) Noise-free model

(b) Noisy model

Figure 5.5: Probability that $N = 2$ given 10 observations (5 blue, 5 green) in (a) the noise-free case and (b) the noisy case. Solid line indicates exact value; '+'s are values computed by 5 sampling runs at intervals of 100,000 samples.

different evidence: that all 10 of the balls that were drawn appeared blue. We use the noisy-observation version of the BLOG model. Figure 5.6(a) shows that when the prior for the number of balls is uniform over $\{1, \ldots, 8\}$, the posterior puts more weight on small numbers of balls; this makes sense because the more balls there are in the urn, the less likely it is that they are all blue. Figure 5.6(b), using a Poisson(6) prior, shows a similar but less pronounced effect.

Note that in Figure 5.6, the posterior probabilities computed by the likelihood weighting algorithm are very close to the exact values (computed by exhaustive enumeration of possible worlds with up to 170 balls). We were able to obtain this level of accuracy using runs of 20,000 samples with the uniform prior, and 100,000 samples using the Poisson prior. On a Linux workstation with a 3.2 GHz Pentium 4 processor, the runs with the uniform prior took about 35 seconds (571 samples/second), and those with the Poisson prior took about 170 seconds (588 samples/second). Such results could not be obtained using any algorithm that constructed a single fixed



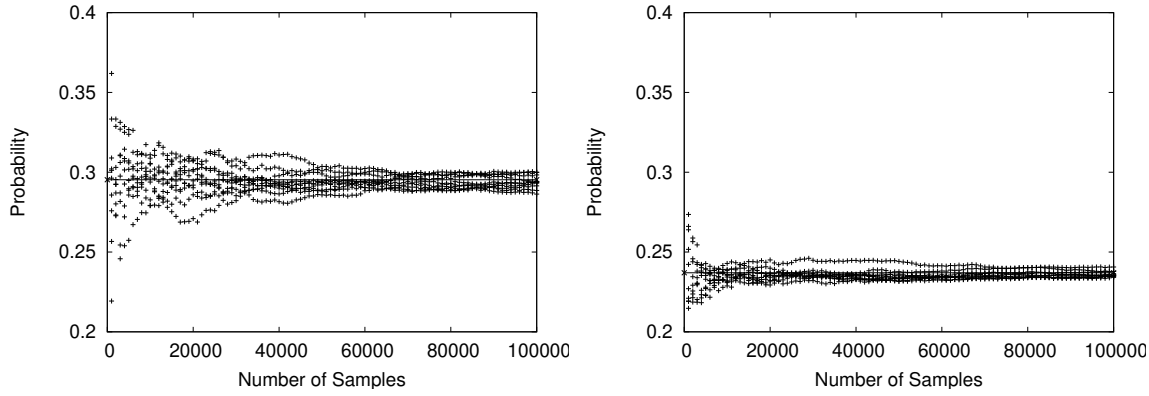
(a) Uniform prior (b) Poisson prior

Figure 5.6: Distribution for the number of balls in the urn (Example 3.1) given noisy observations and two different priors. Dashed lines are the prior; solid lines are the exact posterior given that 10 balls were drawn and all appeared blue; and plus signs are posterior probabilities computed by 5 independent runs of (a) 20,000 or (b) 100,000 samples.

BN, since the number of potentially relevant $V_{\mathsf{TrueColor}}[b]$ variables is infinite in the Poisson case.

Identity uncertainty. In the second experiment, three balls are drawn from the urn: a blue one and then two green ones. We wish to find the probability that the second and third draws produced the same ball. The prior distribution over the number of balls is Poisson(6). Unlike the previous experiment, each ball is blue with probability 0.3.

We ran five independent trials of 100,000 samples on the deterministic and noisy observation models. Fig. 5.7 shows the estimates from all five trials approaching the true probability as the number of samples increases. Note that again, the approximations for the noisy observation model converge more quickly. The noise-free case stays within 0.01 of the true probability after 70,000 samples, while the noisy case



(a) Noise-free model

(b) Noisy model

Figure 5.7: Probability that draws two and three produced the same ball for (a) noise-free observations and (b) noisy observations. Solid line indicates exact value; '+'s are values computed by 5 sampling runs.

converges within 10,000 samples. Thus, we perform inference over a model with an unbounded number of objects and get reasonable approximations in finite time.

5.4 Markov chain Monte Carlo

While likelihood weighting is efficient enough to yield results on some small problems, its convergence becomes extremely slow if there are many hidden variables whose values all have to be set consistently with the evidence. As we observed in Section 2.5.3, Markov chain Monte Carlo (MCMC) methods are a much better option for problems with many hidden variables. In an MCMC algorithm, each sample is generated by conditioning on the previous one. Thus, the sampler can gradually move toward areas of the outcome space with high posterior probability, rather than attempting to find a high-probability area independently on each sample.

In fact, MCMC algorithms have been used to achieve state-of-the-art results on

several real-world problems involving unknown objects. Pasula *et al.* [2003] apply an MCMC algorithm to a citation matching task like the one in Example 4.1, and Oh *et al.* [2004] apply MCMC to a multitarget tracking scenario similar to our Example 4.2. Both of these applications use the Metropolis-Hastings (M-H) algorithm that we discussed in Section 2.5.3. The recent successes of Metropolis-Hastings suggest that it could be applied usefully to other relational inference tasks, such as resolving coreference among noun phrases in text. However, to recycle a comment that Gilks *et al.* made about Gibbs sampling in 1994, "Until now all existing implementations have involved writing one-off computer code in low or intermediate level languages such as C or Fortran." Although perhaps Matlab has replaced Fortran, this is still true about implementations of non-Gibbs-sampling[3] M-H algorithms today: the data structures that represent MCMC states and the algorithms that compute acceptance probabilities are application-specific. Tackling a new application — or even adding a variable to an existing model — requires rewriting portions of the state representation and acceptance probability code, in addition to modifying the proposal distribution.

We would prefer to have a general approximate inference system that computed answers to queries based on a user-supplied probability model and proposal distribution. We already have a formal language, namely BLOG, for representing the models. But we still must determine how to represent MCMC states and compute acceptance probabilities in a way that is flexible enough to support a variety of applications, yet exploits enough problem-specific structure to run at a reasonable speed. We address these questions in this section.

In Section 5.4.1, we discuss the state space of our generic M-H algorithm. The key point here is that it is impractical to use MCMC states that correspond to single possible worlds; instead, states are represented with *partial* descriptions that

[3]The BUGS system, developed by Gilks and his colleagues [Thomas *et al.*, 1992; Gilks *et al.*, 1994], has made Gibbs sampling much easier to use.

denote whole sets of worlds. We provide conditions under which MCMC over sets of worlds yields asymptotically correct answers to queries. Taking advantage of this theorem, we use state descriptions that are partial in two ways: they do not instantiate irrelevant variables, and they abstract away the numbering of interchangeable objects.

Section 5.4.2 discusses the data structures and algorithms necessary to make generic M-H efficient. The goal here is to avoid having the time required to compute the acceptance probability and update the MCMC state grow with the number of hypothesized objects or the number of instantiated variables. We use data structures that represent a proposed world as a set of differences with respect to the current world (Section 5.4.2.1). More interestingly, we can determine which factors in the acceptance probability need to be recomputed by maintaining a Bayes net over the instantiated variables (Section 5.4.2.2). Section 5.5.4 presents experimental results on the citation matching task, showing that our generic M-H system supports the same proposal distribution that was used in a hand-coded implementation, and has a running time of the same order of magnitude.

5.4.1 MCMC states

MCMC states serve as the interface between the general-purpose and application-specific parts of a generic M-H system. The application-specific portions are an initial state distribution q_0, which generates an MCMC state, and a proposal distribution q, which takes in an MCMC state x_n and returns another MCMC state x', along with the proposal probability ratio $q(x_n|x')/q(x'|x_n)$.

The obvious way to apply MCMC to a BLOG model is to let the MCMC states be possible worlds. However, proposal distributions that are used in practice often do not propose complete partial worlds. For instance, the proposer used by [Pasula

et al., 2003] uses moves that split and merge publications. It only proposes titles and author names for publications that are cited by some citation in the proposed state. By contrast, a full possible world for a model such as the one in Figure 4.3 typically contains many publications that are not cited, that is, are not the value of PubCited(c) for any citation c. The world must specify the values of the Title and NthAuthor functions on all the publications that exist, including the uncited ones.

In fact, in our model for citations, the attributes of uncited publications are irrelevant for inference: in a world where a publication p is uncited, the Title and NthAuthor variables defined on p are not active ancestors of query or evidence variables. Proposing values for these variables would be a waste of time. In fact, in some BLOG models — such as a model for aircraft tracking with variables State(a, t) for every aircraft a and natural number t — each possible world assigns non-null values to infinitely many variables. In such models, proposing and storing full possible worlds would require infinite time and space.

5.4.1.1 Events as MCMC states

Our generic MCMC architecture circumvents these difficulties by allowing proposal distributions to use *partial* descriptions of possible worlds. For instance, the proposer for citation matching specifies the values of the PubCited function on all citations, and specifies attributes for the cited publications and their authors. Such a partial specification can be thought of as an event: a set of full possible worlds that satisfy the specification.

Thus, our system runs a Markov chain over a set Σ of *events*, which are subsets of the outcome space Ω. The following theorem gives conditions under which a Markov chain over Σ will yield correct answers to queries.

Theorem 5.11. *Let P be a probability distribution over a set Ω, E and Q be subsets*

of Ω, and Σ be a set of subsets of Ω. Suppose $s_1, s_2, \ldots, s_N$ are samples from an ergodic Markov chain over Σ with stationary distribution proportional to $P(s)$. If:

1. *Σ is a partition of E; and*

2. *for each $s \in \Sigma$, either $s \subseteq Q$ or $s \cap Q = \varnothing$,*

then $\frac{1}{N}\sum_{n=1}^{N} \mathbf{1}_{s_n \subseteq Q}$ converges to $p(Q|E)$.

Proof. Let π be the stationary distribution of the Markov chain, and let $\widetilde{Q} = \{s \in \Sigma : s \subseteq Q\}$. Then by standard results about ergodic Markov chains, $\frac{1}{N}\sum_{n=1}^{N} \mathbf{1}_{s_n \subseteq Q}$ converges to $\pi(\widetilde{Q})$ as $N \to \infty$. So it suffices to show that $\pi(\widetilde{Q}) = p(Q|E)$. By definition, $\pi(\widetilde{Q}) = \sum_{s \in \widetilde{Q}} \pi(s)$. Then since $\pi(s)$ is defined on Σ and is proportional to $p(s)$:

$$\pi(\widetilde{Q}) = \frac{\sum_{s \in \widetilde{Q}} P(s)}{\sum_{s \in \Sigma} P(s)} \tag{5.1}$$

By the assumption that Σ is a partition of E, we know $\sum_{s \in \Sigma} P(s) = P(E)$. We now argue that $\widetilde{Q}$ is a partition of $Q \cap E$. To see this, consider any $\omega \in Q \cap E$. Because Σ is a partition of E, there is exactly one set $s \in \Sigma$ such that $\omega \in s$. Given that $\omega \in s \cap Q$, it follows by assumption 2 that $s \subseteq Q$. Therefore $s \in \widetilde{Q}$. Thus, since $\widetilde{Q} \subseteq \Sigma$, there is exactly one $s \in \widetilde{Q}$ containing ω. So $\widetilde{Q}$ is a partition of $Q \cap E$ and $\sum_{s \in \widetilde{Q}} P(s) = P(Q \cap E)$. Plugging into Eq. 5.1, we find that $\pi(\widetilde{Q}) = \frac{P(Q \cap E)}{P(E)} = P(Q|E)$. □

The next section discusses a way to choose the event set Σ.

5.4.1.2 Partial instantiations

The most straightforward events to use as MCMC states are those corresponding to partial instantiations of the basic random variables. To satisfy Theorem5.11, these partial instantiations must instantiate the evidence variables to their observed values,

instantiate the query variables, and define a partition of the worlds consistent with the evidence.

Furthermore, to compute the acceptance probability given in Eq. 2.10, the system must be able to compute the ratio $P(s')/P(s_n)$ for events $s_n, s' \in \Sigma$. In general, it is not easy to compute the probability of a partial instantiation: for instance, if the instantiation just includes the evidence variables, then computing its probability involves summing out all the hidden variables. In some cases it is possible to sum out uninstantiated variables analytically, but our generic MCMC system currently cannot do so.

Instead, we limit ourselves to partial instantiations whose probabilities are given by simple product expressions. These are the *self-supporting* instantiations: those that include all the active parents of the variables they instantiate. To say this formally, we need a bit more background on contingent BNs (see Chapter 3 for details). In a contingent BN, the conditional probability distribution (CPD) for a variable V is given by a tree where each internal node is labeled with a parent variable U, edges out of a node are labeled with values of U, and each leaf is labeled with a probability distribution over V. A particular parent variable may occur on some paths through the tree and not on others: for instance, in the tree for $\mathsf{TitleText}(\mathsf{Cit1})$, the root is labeled with $\mathsf{PubCited}(\mathsf{Cit1})$, and the variable $\mathsf{Title}((\mathsf{Pub}, 7))$ occurs only in the subtree where $\mathsf{PubCited}(\mathsf{Cit1}) = (\mathsf{Pub}, 7)$. An instantiation σ *supports* V if it is complete enough so that only one path through the tree is consistent with σ. This path leads to a leaf with some distribution over V; we write $p_V(v|\sigma)$ for the probability of the value v under this distribution.

An instantiation is *self-supporting* if it supports every variable that it instantiates. By the semantics of a contingent BN, if σ is a finite, self-supporting instantiation,

then:

$$P(\sigma) = \prod_{X \in \text{vars}(\sigma)} c^X(\sigma_X, \lambda^X(\sigma)) \tag{5.2}$$

where σ_X is the value that σ assigns to X. Thus, if we use self-supporting partial instantiations as our MCMC states, we can compute $P(s')/P(s_n)$ with no summations.

To satisfy the conditions of Theorem 5.11, we need to use self-supporting instantiations that form a partition of E. In particular, we need to ensure that these instantiations are mutually exclusive: if some of them define overlapping events, then worlds occurring in several events will be overcounted. The following result ensures that we can avoid overlaps by using "minimal" instantiations.

Definition 5.1. *Let* $\mathbf{V}$ *be a set of random variables, and* σ *be a self-supporting instantiation that instantiates* $\mathbf{V}$*. Then* σ *is* minimal beyond $\mathbf{V}$ *if no sub-instantiation of* σ *that instantiates* $\mathbf{V}$ *is self-supporting.*

Proposition 5.12. *Let* $\mathbf{V}$ *be a set of random variables in a contingent BN. The self-supporting instantiations that are minimal beyond* $\mathbf{V}$ *are mutually contradictory.*

Proof. Assume for contradiction that two distinct self-supporting instantiations σ and τ that are minimal beyond $\mathbf{V}$ are both satisfied by some world ω. By definition, neither σ nor τ can be a sub-instantiation of the other. Therefore σ instantiates a variable, call it X^*, that τ does not instantiate. Consider a graph over vars (σ) where there is an edge from X to Y if the path through Y's CPD tree that is consistent with ω contains a node labeled with X. Since σ is minimal beyond $\mathbf{V}$, there must be a directed path in this graph from X^* to $\mathbf{V}$; otherwise the sub-instantiation obtained by removing X^* and all its descendents would still instantiate $\mathbf{V}$ and be self-supporting. But since τ is also consistent with ω, τ must instantiate all the variables along this directed path in order to be self-supporting. This contradicts

the assumption that τ does not instantiate X^*. □

We have now identified a set of partial instantiations that satisfy the conditions of Thm. 5.11 and have probabilities that are easy to compute. If the evidence variables are $\mathbf{V}_E$ and the query variables are $\mathbf{V}_Q$, we use the set of self-supporting instantiations that assign the observed values to $\mathbf{V}_E$ and are miminal beyond $\mathbf{V}_E \cup \mathbf{V}_Q$.

5.4.1.3 Object identifiers

Recall that in a BLOG model, the objects that satisfy a given number statement are numbered. For instance, in worlds where there are 10 publications, the publication objects are $(\mathsf{Pub}, 1), \ldots, (\mathsf{Pub}, 10)$. We will refer to guaranteed and non-guaranteed objects that exist in possible worlds as *concrete objects*. But the split-merge proposal distributions used, for example, by Pasula *et al.* [2003] do not make any provisions for numbering the publications they create. When the proposal takes a publication, say $(\mathsf{Pub}, 7)$, and splits off some co-referring citations to join a new publication, what number does this new publication get? In choosing a numbering scheme, we have to ensure that merge moves — for example, eliminating $(\mathsf{Pub}, 3)$ and adding all its citations to $(\mathsf{Pub}, 7)$ — are still reversible. That is, the probability of splitting $(\mathsf{Pub}, 7)$ and getting $(\mathsf{Pub}, 3)$ back again must be nonzero. One solution is to choose the new publication's number randomly from the numbers that are unused in the current partial instantiation. But then the proposer must take these numbering choices into account when computing forward and backward proposal probabilities.

All this bother about numbering seems unnecessary, since the publication objects are interchangeable. In fact, our general MCMC system includes an additional layer of abstraction that can eliminate the need to worry about numbering when writing proposal distributions. The idea is to specify MCMC states using *abstract* partial

instantiations, in which unnumbered *object identifiers* can be used as both arguments and values for basic random variables. For instance, an abstract partial instantiation could say: PubCited(Cit1) = Pub@A3F, Title(Pub@A3F) = "foo".

Definition 5.2. *An* abstract function application variable *has the form* $A_f[o_1, \ldots, o_k]$ *where* f *is a* k*-ary function symbol and* $o_1, \ldots, o_k$ *are concrete objects or object identifiers. An* abstract partial instantiation σ *consists of a set of number variables*[4] *and abstract function application variables, denoted* $\text{vars}(\sigma)$*, and a function that maps each element of* $\text{vars}(\sigma)$ *to a concrete object or object identifier. For each type, an abstract partial instantiation uses either object identifiers or concrete objects to represent the non-guaranteed objects, not both.*

Semantically, object identifiers can be thought of as existentially quantified logical variables. The abstract partial instantiation used as an example above is equivalent to $\exists x((\text{PubCited}(\text{Cit1}) = x) \wedge (\text{Title}(x) = \text{"foo"}))$. When an abstract instantiation uses several object identifiers, they are also asserted to be distinct.

Definition 5.3. *A partial instantiation* τ *is a* concrete version *of an abstract partial instantiation* σ *if there is a one-to-one function* h *from object identifiers used in* σ *to concrete objects that exist in some world consistent with* τ*, such that* σ *instantiates* $A_f[o_1, \ldots, o_k]$ *if and only if* τ *instantiates* $V_f[h(o_1), \ldots, h(o_k)]$*, and* $h(\sigma(A_f[o_1, \ldots, o_k])) = \tau(V_f[h(o_1), \ldots, h(o_k)])$*. A world* satisfies *an abstract partial instantiation* σ *if and only if it satisfies some concrete version of* σ*.*

For instance, the abstract partial instantiation:
#Pub = 3, PubCited(Cit1) = Pub@A3F, Tit(Pub@A3F) = "foo"
has three concrete versions:
#Pub = 3, PubCited(Cit1) = (Pub, 1), Tit((Pub, 1)) = "foo"

[4] In models where objects generate other objects, number variables can also be abstract.

#Pub = 3, PubCited(Cit1) = (Pub, 2), Tit((Pub, 2)) = "foo"

#Pub = 3, PubCited(Cit1) = (Pub, 3), Tit((Pub, 3)) = "foo"

Each abstract instantiation corresponds to an event, namely the set of possible worlds that satisfy it. If Σ is a set of such events that obeys the conditions of Thm. 5.11, then we can use Σ as our state space for MCMC. However, just as two formulas that use different variable names may be satisfied by the same set of possible worlds, two abstract instantiations with different object identifiers can represent the same event. Thus, we must keep in mind that we are running MCMC over events, not over abstract partial instantiations. In particular, the proposal probability $q(s'|s_n)$ is the probability of one event given another; it cannot depend on how the event s_n is represented.

But how do we compute the probability of the event corresponding to an abstract instantiation σ? Def. 5.2 stipulates that if an abstract instantiation uses object identifiers for a type, it does not also use non-guaranteed concrete objects of that type. This implies that the concrete versions of an abstract instantiation σ form an isomorphism class: we can change one into another by permuting the non-guaranteed objects. So all the concrete versions have the same probability, which we will denote $p_c(\sigma)$. If the concrete versions are self-supporting, then $p_c(\sigma)$ can be computed using a product expression. The question is, how many concrete versions does a given abstract instantiation have?

If an abstract instantiation σ contains an instantiated number variable asserting that there are n objects of a given type, and σ uses m object identifiers of that type, then there are ${}_nP_m \triangleq \frac{n!}{(n-m)!}$ distinct functions that could play the role of h in Def. 5.3. However, this does not always mean that σ has ${}_nP_m$ distinct concrete versions. For instance, suppose σ simply says: #Pub = 10, Title(Pub@A3F) = "foo", Title(Pub@B46) = "foo". Here an h function that maps Pub@A3F to $(\mathsf{Pub}, 1)$ and Pub@B46 to $(\mathsf{Pub}, 2)$ yields the same concrete instantiation as one that does the

opposite, since σ makes the same assertion about Pub@A3F and Pub@B46. Thus, the number of distinct concrete versions is only $\frac{1}{2}\,({}_{10}P_2)$.

The difficulty in this example is that σ has a non-trivial automorphism: interchanging Pub@A3F and Pub@B46 yields σ itself. In general, the number of distinct concrete versions of an abstract instantiation with a automorphisms is $\frac{1}{a}\,({}_nP_m)$. But if the instantiation specifies relations among the non-guaranteed objects — for instance, publications citing one another — then counting automorphisms becomes difficult. Indeed, counting the number of automorphisms of an undirected graph is polynomially equivalent to determining whether two graphs are isomorphic [Mathon, 1979], a problem for which no polynomial-time algorithm is known. This issue of automorphisms does not just arise because we are trying to use abstract partial instantiations as MCMC states: if we required the proposal distribution to assign numbers to objects, its proposal probability calculations would need to determine how many different numberings would yield the same proposal.

Fortunately, for many models of practical interest, there is a simple way to avoid this issue. The abstract partial instantiations that we use for citation matching only make assertions about cited publications. So if σ uses an object identifier i, then σ asserts $\mathsf{PubCited}(c) = i$ for some citation c. Two h functions that map i to different concrete objects cannot yield the same concrete instantiation, because they yield different values for $\mathsf{PubCited}(c)$. In general:

Definition 5.4. *An object identifier i is* grounded *in an abstract partial instantiation σ if there is a logical ground term t_i such that every mapping function h (as in Def. 5.3) yields a concrete instantiation where $h(i)$ is the value of t_i.*

Proposition 5.13. *Suppose σ is an abstract partial instantiation whose concrete versions are self-supporting instantiations having probability $p_c(\sigma)$. Let T be the set of types for which σ uses identifiers, and assume that for every type $\tau \in T$,*

σ instantiates a number variable asserting that there are n_τ non-guaranteed objects of type τ. Let s be the event corresponding to σ. If every identifier used in σ is grounded, then:

$$P(s) = P_c(\sigma) \prod_{\tau \in T} {}_{n_\tau} P_{m_\tau} \tag{5.3}$$

where m_τ is the number of identifiers of type τ used in σ.[5]

Logical ground terms include not just expressions such as PubCited(Cit1), but also nested expressions such as NthAuthor(PubCited(Cit1), 1). The requirement that object identifiers be grounded is not burdensome in scenarios — such as citation matching — where the relevant objects are those connected to guaranteed objects by some chains of function applications. In BLOG models that involve weighted sampling or aggregation, non-guaranteed objects that do not serve as function values may become relevant. In such cases, the proposer would need to represent such objects concretely.

In cases where Prop. 5.13 applies, we can compute the probability of an abstract instantiation by just computing the probability of one of its concrete versions and then multiplying in an adjustment factor that is a product of factorials. In fact, the ratio of these adjustment factors in the acceptance probability is the same as the ratio of adjustments to the backward and forward proposal probabilities that would emerge if we required the proposal distribution to choose numbers for objects. But we have shifted this computation out of the application-specific proposal distribution and into general-purpose code.

[5]This result can be extended to cases where objects generate objects; then the product is not over types, but over applications of number statements to tuples of generating objects. Abstract instantiations must be extended to specify the generating objects for each object identifier.

5.4.2 Performing M-H steps efficiently

Our overall goal is to compute the probability ratio and update the MCMC state in time that does not grow with the number of existing objects or the number of instantiated variables. This is not always possible, but application-specific implementations exploit various forms of structure to do these computations in constant time. We are able to exploit some of the same structure in our generic system.

5.4.2.1 Difference data structures

We have said that the proposal distribution takes in an MCMC state s_n and returns a new state s'. Constructing s' from scratch would take time linear in the number of instantiated variables. However, proposals typically change only a small fraction of the variables. Thus, our implementation represents s' as a difference structure or "patch" relative to the current state s_n. This difference structure supports all the same access methods as the original state: if a client asks for the value of a variable that has not been changed, the request is just passed through to the original state. Constructing s' takes time that depends only on the number of changed variables.

If the proposal is rejected, the patch is simply discarded, and s_{n+1} is set equal to s_n. If the proposal is accepted, then s_{n+1} is obtained by applying the patch to s_n, which again takes time linear in the number of changed variables.

5.4.2.2 Computing the acceptance probability

Besides maintaining the MCMC state, the main task for our general-purpose code is to compute the acceptance probability. The proposal distribution provides $q(s_n|s')/q(s'|s_n)$, so we must compute the probability ratio $P(s')/P(s_n)$. If s_n and s' are represented

as self-supporting partial instantiations σ_n and σ', Eq. 5.2 tells us that this ratio is:

$$\frac{P(\sigma')}{P(\sigma_n)} = \frac{\prod_{V \in \text{vars}(\sigma')} c^V(\sigma'_V|\sigma')}{\prod_{V \in \text{vars}(\sigma_n)} c^V((\sigma_n)_V|\sigma_n)} \tag{5.4}$$

Computing this ratio naively would require time proportional to the number of instantiated variables in σ' and σ_n. But fortunately, many of the factors in the numerator and denominator may cancel.

Definition 5.5. *If a partial instantiation σ supports a variable V, then the* active parents *of V in σ are those variables that occur as labels on nodes in V's CPD tree on the unique path that is consistent with σ.*

Proposition 5.14. *Suppose two partial instantiations σ and σ' agree on a variable V and all the variables that are active parents of V in σ. Then $p_V(\sigma'(V)|\sigma') = p_V(\sigma(V)|\sigma)$. Also, V has the same active parents in σ' as in σ.*

Thus, we only need to compute the factors for variables that are newly instantiated, uninstantiated, or changed in σ', or whose parents have changed values. Because we are explicitly representing the differences between σ' and σ_n (see Section 5.4.2.1), we can identify changed variables efficiently.

However, it is not so easy to identify variables whose active parents have changed. We can enumerate V's active parents in σ_n by walking through V's CPD tree. But if only a few variables have changed, we don't want to iterate over all variables, seeing which ones happen to have a changed variable as an active parent.

To avoid this iteration, we maintain a graph over the instantiated variables, containing those edges that are active given σ_n. Each variable V has pointers to its children, that is, the variables of which it is an active parent in σ_n. Given this data structure, we can efficiently enumerate the children of all variables that are changed in σ'. The BN is constructed on the initial state, and then updated

after each accepted proposal. Conveniently, by Prop. 5.14, it suffices to update the active parent sets for variables whose parent values have changed — which we are enumerating anyway to recompute their probability factors.

If we use abstract partial instantiations, the probability ratio includes the factorial adjustment factors given in Eq. 5.3. Again, computing these factorials naively would take time linear in the magnitudes of the number variables. But if the proposal makes small changes to the values of number variables and the number of used identifiers, then most of the factors inside the factorials cancel out.

The calculation techniques presented here do have some limitations. One is that a variable's child set may grow linearly with the number of objects. In the citation matching model, where the probability that a PubCited variable takes on any particular value in a world with N publications is $1/N$, the #Pub variable is always an active parent of all PubCited variables. So the time required to compute the acceptance probability for a proposal that changes the number of publications grows linearly with the number of citations. This slowdown could be avoided by recognizing that every PubCited variable makes the same contribution the probability ratio, so we can compute this contribution once and raise it to the power of the number of citations. However, our current implementation does not detect when this can be done. Conversely, a variable's active parent set may grow linearly with the number of hypothesized objects: this happens in cases of weighted sampling or aggregation. Finally, our approach does not allow the system to detect cancellations between the $P(s')$ and $q(s'|s_n)$ factors, such as occur in Gibbs sampling [Gelman, 1992].

5.5 Application to citation matching

In this section, we discuss an application of the BLOG MCMC framework to a real-world task: taking citations from the reference lists of online papers, and deter-

mining which ones refer to the same publications. Our research group developed a probabilistic model and MCMC proposal distribution for this *citation matching* task before we began developing BLOG [Pasula *et al.*, 2003]. We shall evaluate how easy it is to describe this model in BLOG; how well our MCMC framework supports the previously developed proposal distribution; and how much speed we sacrifice in using a general-purpose inference engine rather than an inference program hand-coded for this application.

5.5.1 Citation matching

In the 1990s, as large numbers of scientists began making their papers available online, the developers of the CiteSeer system [Giles *et al.*, 1998] proposed to automatically construct a database of scientific literature using the citations at the ends of these papers. An important challenge in constructing such a system is *citation matching* [Lawrence *et al.*, 1999b]: determining when two citations, written by different authors in different formats, refer to the same publication.[6] Identifying coreferent citations is important for avoiding duplicates when listing the papers by a particular author; for listing all the places where a particular paper is cited; and for constructing full bibliographic records (including page numbers, authors' full names, etc.) based on information in several citations.

Telling when two citations are coreferent is more challenging than it may seem. Figure 5.8 shows three citations that look quite different, but in fact refer to the same paper. One difficulty is the diversity of citation formats: for instance, the second citation in Figure 5.8 puts the title before the author list, while the other two citations do the reverse. Even distinguishing the fields in a given citation can be

[6] Another problem is determining when a particular PDF or PostScript file contains the paper referred to by a set of citations; we do not address this problem here, although our model could be extended to handle it.

```
[9] Lashkari, Yezdi, Metral, Max, and Maes Pattie, Collaborative
Interface Agents, Proceedings of the Twelfth National Conference
on Artificial Intelligence, MIT Press, Cambridge, MA, 1994.

[Lashkari et al 94]  Collaborative Interface Agents, Yezdi Lashkari,
Max Metral, and Pattie Maes, Proceedings of the Twelfth National
Conference on Articial Intelligence, MIT Press, Cambridge, MA,
1994.

Metral M. Lashkari, Y. and P. Maes. Collaborative interface agents.
In Conference of the American Association for Artificial
Intelligence, Seattle, WA, August 1994.
```

Figure 5.8: Three citations that refer to the same publication.

difficult. In the second citation, if one did not realize that "Collaborative Interface Agents" is much more likely to be a title than an author name, one could interpret "Collaborative Interface Agents, Yezdi Lashkari, Max Metral, and Pattie Maes" as an author list, and "Proceedings of the Twelfth National Conference on Articial Intelligence" as the title of the publication being cited. Author names are written in many different formats — in Figure 5.8 we see "Maes Pattie", "Pattie Maes" and "P. Maes". All parts of a citation are subject to typographical errors and PostScript extraction problems, as seen in "Articial" in the second citation. Even in the absence of errors and formatting differences, people use different names to refer to conferences ("National Conference on Artificial Intelligence" versus "Conference of the American Association for Artificial Intelligence") and mention objects whose relation to the cited publication may be ambiguous (in Figure 5.8, "Cambridge, MA" is the place where the proceedings were published, while "Seattle, WA" is the place where the conference occurred).

Lawrence *et al.* [1999b] experiment with several methods for matching citations, and achieve the best accuracy with one they call "Word and Phrase Matching".

Roughly speaking, this algorithm clusters citations based on how many words and phrases they have in common, where a "phrase" is a sequence of two consecutive words. More details are given in their paper.

Lawrence *et al.* also introduce a methodology for evaluating the accuracy of a citation matching system. The first step is to have humans identify the clusters of co-referring citations in a sample of citations extracted from online papers. A random sample might not yield enough co-referring papers to be interesting, so Lawrence *et al.* use citation sets obtained by searching a large collection for a certain substring, such as "face" or "constraint". Then, to evaluate a given algorithm, Lawrence *et al.* compute the fraction of true (human-identified) clusters that the system recovers exactly. Wellner *et al.* [2004] refer to this metric as *cluster recall*.

5.5.2 BLOG model

We gave a simple model for citation matching in Chapter 4 (Figure 4.3). That model represents three kinds of objects: citations, publications, and researchers. The priors over researcher names and titles are defined by elementary CPDs called `NamePrior` and `TitlePrior`, and the conditional distribution over citation strings given the underlying titles and author names is defined by `NoisyCitationGrammar`. The model used by Pasula *et al.* [2003] can be seen as an instance of this one, with particular implementations of these CPDs. However, the details of these CPDs are implicit in the Lisp code used for the Pasula *et al.* paper; we do not attempt to reproduce them exactly in our BLOG implementation. Instead, we use essentially the same model as an application-specific Java implementation that we developed in the summer of 2003. This Java implementation serves as our reference for comparison. It is worth noting that all these models are quite incomplete: they do not represent all the entities that are mentioned in citation strings, such as journals, conferences,

```
1   type Researcher;
2   #Researcher ~ RoundedLogNormal[100, 1];

3   random String Surname(Researcher r)
4       ~ CharNgram["model/surname"];
5   random String GivenName(Researcher r)
6       ~ GivenNameDistrib["model/given"];

7   type Publication;
8   #Publication ~ RoundedLogNormal[1000, 1];

9   random String Title(Publication p)
10      ~ CharNgram["model/Title"];
11  random NaturalNum NumAuthors(Publication p)
12      ~ NatNumDistribWithTail[[0.01; 0.19; 0.5; 0.2; 0.1],
13                              0.8, 0.1];
14  random Researcher NthAuthor(Publication p, NaturalNum n)
15      if n < NumAuthors(p) then
16          ~ UniformChoice({Researcher r:  !exists NaturalNum m
17                           ((m < n) & (NthAuthor(p, m) = r))});
```

Figure 5.9: The beginning of the BLOG model used in our application.

publishers, and cities. We plan to develop BLOG models that incorporate these additional objects in future work.

Looking at the BLOG model in Figure 4.3 provides little information about the probability distributions for titles, author names, and citation strings; these details are hidden inside elementary CPDs. In our experiments, we use a BLOG model that makes this information more explicit and uses simpler elementary CPDs. This explicitness also makes the model significantly longer than the one in Figure 4.3, so we present it in several parts.

Figure 5.9 shows the beginning of the model, describing researchers and publications and their attributes. Note that all the dependency statements in this model

also serve as function declarations, as discussed at the end of Section 4.2.5. The prior distributions for the numbers of researchers and publications are given by `RoundedLogNormal` CPDs, which define very flat and broad distributions over natural numbers. A real-valued random variable X has a *log-normal* distribution if $\ln(X)$ has a normal distribution. Under a *rounded log-normal* distribution, the probability of a natural number n is the probability of the interval $[n - 0.5, n + 0.5]$ under a log-normal distribution, which is the probability of $[\ln(n - 0.5), \ln(n + 0.5)]$ under the underlying normal distribution.[7] The parameters of a `RoundedLogNormal` CPD are the mean of the desired distribution, and the variance of the underlying normal distribution. Thus, `RoundedLogNormal[100, 1]` yields a distribution whose mean is 100, and that assigns about 95% of its probability mass to numbers in the range $[100e^{-2}, 100e^{2}] \approx [14, 738]$.

As shown in Figure 5.9, we model the surnames and given names of researchers separately. The CPD for Surname is a character n-gram model trained on the surnames of authors appearing in a large BibTeX file from the University of Pennsylvania. [8] It interpolates between unigram, bigram and trigram models using weights estimated on held-out data [Bahl *et al.*, 1983].

GivenName(r) actually represents researcher r's whole given name string, which may consist of several names and initials separated by spaces. The elementary CPD `GivenNameDistrib` defines a distribution over strings obtained by first choosing how many given names to include (according to a distribution specified internally), then generating each given name from a character n-gram model, and concatenating the names with spaces in between. The character n-gram model used here is trained on

[7] As an approximation, rather than integrating the normal density over the interval $[\ln(n - 0.5), \ln(n + 0.5)]$, we just take the value of the normal density at $\ln(n)$ and multiply it by $\ln(n + 0.5) - \ln(n - 0.5)$.

[8] This file contains about 10,500 BibTeX entries and is available at http://liinwww.ira.uka.de/bibliography/Ai/pennbib.html.

given names and initials of authors in the U. Penn BibTeX file. The model does not distinguish between full given names and initials: it just treats initials as names that happen to consist of a single character. This means that if all the observed references to a particular researcher use the given name string "James K.", the model will assign high posterior probability to worlds where this researcher's GivenName value is "James K". The inference algorithm will not be forced to hypothesize worlds where the "K" is expanded to some longer name. Of course, given names that are written in full in a researcher's GivenName string may be shortened to initials in some citations; this phenomenon is handled later in the model.

The distribution for the title of a publication is also given by a character n-gram model trained using the U. Penn BibTeX file. For the number of authors on a publication, the elementary CPD is `NatNumDistribWithTail`, which defines a mixture between an explicitly specified distribution over the first few natural numbers and a geometric distribution over the remaining numbers. For instance, the CPD on line 12 of Figure 5.9 specifies explicit probabilities for the numbers zero through four; this distribution gets a mixture weight of 0.8, and is mixed with a geometric distribution with success probability 0.1 over the numbers greater than four. Now for NthAuthor(p, n), we use a set expression to implement sampling without replacement from the set of researchers. The formula in this set expression enforces the fact that the nth researcher in an author list cannot be the same one that appears at an index $m < n$.

We can now move on to the part of the model that deals with citations. As shown in Figure 5.10, we declare a type Citation with a certain number of guaranteed objects. The number of guaranteed citation objects (set to 349 in the figure) is the number of citation strings we wish to reason about in a particular run.[9] The

[9]The BLOG inference engine actually allows us to specify statements that are specific to a particular run — such as this guaranteed object statement — separately from the main model file. We include the statement in the model here for clarity.

```
1   type Citation;
2   guaranteed Citation Cit[349];

3   random Publication PubCited(Citation c)
4       ~ UniformChoice(Publication p);

5   random String TitleAsCited(Citation c)
6       ~ StringEditModel(Title(PubCited(c)));

7   random NaturalNum NumAuthorsDropped(Citation c)
8       ~ Binomial[0.005](NumAuthors(PubCited(c)));
9   random NaturalNum NumAuthorsAdded(Citation c)
10      if (NumAuthorsDropped(c) = 0) then ~ Geometric[0.001]
11      else = 0;
12  random NaturalNum NumAuthorsListed(Citation c)
13      = NumAuthors(PubCited(c))
14            - NumAuthorsDropped(c) + NumAuthorsAdded(c);
```

Figure 5.10: Portion of the BLOG model for citation matching that deals with the titles and author lists that appear in citations.

publication cited by each citation is chosen uniformly from the set of citations. Next, we introduce a string-valued function TitleAsCited(c), whose value is derived from Title(PubCited(c)) according to the `StringEditModel` CPD. This elementary CPD models typographical errors, using a probabilistic form of edit distance. The error model is based on a generative process where characters are deleted, inserted, or substituted for one another with certain probabilities. The probability of an output string s given an input string t is the sum of the probabilities of the sequences of operations that generate s from t; we use an algorithm of Bahl and Jelinek [1975] to compute such probabilities in $O(|s| \times |t|)$ time. The probabilities of the individual insertion, deletion, and substitution operations are set by hand.

The remaining statements in Figure 5.10 model the process by which people add and drop authors when writing a citation. We use a simple model in which the cita-

tion writer either drops or adds authors, but does not do both; this means that the number of authors dropped or added can be found simply by comparing the lengths of the author lists in the citation and its underlying publication. Lines 7–8 specify that the number of authors dropped has a binomial distribution, equivalent to that obtained by dropping each author independently with probability 0.005. If no authors are dropped, then line 10 says that the number added has a geometric distribution with success probability 0.001. Again, these probabilities are set by hand. The last few lines in Figure 5.10 introduce a function NumAuthorsListed(c) whose value is derived deterministically from NumAuthors(PubCited(c)), NumAuthorsDropped(c) and NumAuthorsAdded(c) using the built-in functions "+" and "−" (see Table 4.2). This function is introduced solely to make subsequent statements simpler.

The next thing to model is how the text for each author name is generated in the citation. Specifically, we introduce string-valued function symbols SurnameAsCited and GivenNameAsCited, as well as a Boolean function symbol NameReversed specifying whether the surname is written first, and a string-valued function symbol InternalSep specifying the characters that come after the surname if the name is reversed. We could index all these functions by citations c and natural numbers n, and let them yield null values for $n >$ NumAuthorsListed(c). But this would be somewhat cumbersome. It is more intuitive to think of each citation as containing a sequence of author mentions, with the random function symbols we have just discussed being defined on AuthorMention objects.

Figure 5.11 shows how we implement this approach in our BLOG model. The type AuthorMention has two origin functions, Container (yielding a citation) and Index (yielding a natural number). AuthorMention has one number statement (lines 4–5), specifying that the number of AuthorMention objects with container c and index n is one if $n <$ NumAuthorsListed(c), and (by default) zero otherwise. Next, we introduce a function symbol Referent that maps author mentions to the researchers

```
1   type AuthorMention;
2   origin Citation Container(AuthorMention);
3   origin NaturalNum Index(AuthorMention);
4   #AuthorMention(Container = c, Index = n)
5       if n < NumAuthorsListed(c) then = 1;

6   random Researcher Referent(AuthorMention m)
7       if Index(m) < NumAuthors(PubCited(Container(m)))
8           then = NthAuthor(PubCited(Container(m)), Index(m));

9   random String SurnameAsCited(AuthorMention m)
10      if Referent(m) = null then ~ CharNgram["model/surname"]
11      else ~ StringEditModel(Surname(Referent(m)));
12  random String GivenNameAsCited(AuthorMention m)
13      if Referent(m) = null then ~ GivenNameDistrib["model/given"]
14      else ~ GivenNameEditModel(GivenName(Referent(m)));

15  random Boolean NameReversed(AuthorMention m)
16      if GivenNameAsCited(m) != "" then ~ Bernoulli[0.5];
17  random Boolean HasComma(AuthorMention m)
18      if NameReversed(m) then ~ Bernoulli[0.9];
19  random Boolean HasCommaSpace(AuthorMention m)
20      if HasComma(m) then ~ Bernoulli[0.9];
21  random String InternalSep(AuthorMention m)
22      if HasCommaSpace(m) then = ", "
23      elseif HasComma(m) then = ","
24      elseif NameReversed(m) then = " ";

25  random String NameListed(AuthorMention m)
26      if NameReversed(m)
27          then = Concat(Concat(SurnameAsCited(m), InternalSep(m)),
28                        GivenNameAsCited(m))
29      elseif GivenNameAsCited(m) != ""
30          then = Concat(Concat(GivenNameAsCited(m), " "),
31                        SurnameAsCited(m))
32      else = SurnameAsCited(m);
```

Figure 5.11: Portion of the BLOG model dealing with author mentions.

they refer to. In a more sophisticated model, the dependency statement for Referent might allow authors to be re-ordered, and allow authors to be dropped and added anywhere in the author list. In this model, however, we assume that if the index of a mention m is a valid index in the underlying publication's author list, then Referent(m) is the author at that index. Otherwise, Referent(m) defaults to null. The SurnameAsCited and GivenNameAsCited values are generated from prior distributions if Referent(m) is null; otherwise, they are generated according to string edit models. GivenNameAsCited actually uses a special CPD `GivenNameEditModel`, which allows names to be shortened to initials (with a certain probability) in addition to the usual typographical errors.

Because we are defining a generative model for citation strings, we also need to specify how the surname and given name strings are combined. NameReversed is a Boolean function symbol that governs whether the surname comes first. If NameReversed(m) is true, then HasComma(m) may be true to indicate that there is a comma after the surname; if HasComma(m) is true, then HasCommaSpace(m) may be true to indicate that there is a space after the comma. These functions determine the string InternalSep(m) — representing the separator between the surname and given names — which then contributes to determining NameListed(m), the name as it actually appears in the citation string.

Thus, we have seen how the text of each author mention is generated. The next step is to construct the full author list, including separators between the author names, as well as any punctuation that marks the end of the list. Figure 5.12 shows the statements the model this process. First, for each citation c and each natural number $n <$ NumAuthorsListed(c), NthMention(c, n) is deterministically equal to the unique AuthorMention object whose container is c and whose index is n. Here we use the `Iota` elementary CPD, which takes a singleton set as an argument and defines a distribution assigning all probability to the sole element of that set.

```
1   random AuthorMention NthMention(Citation c, NaturalNum n)
2       if n < NumAuthorsListed(c)
3           then ~ Iota({AuthorMention m :
4                           Container(m) = c & Index(m) = n});

5   random String NthAuthorSeparator(Citation c, NaturalNum n)
6       if (NumAuthorsListed(c) > 1) & (n < Pred(NumAuthorsListed(c)))
7           then ~ TokenUnigram["model/AuthorSep"];
8   random String AuthorListTerminator(Citation c)
9       ~ TokenUnigram["model/AuthorTermin"];

10  random String FirstNAuthorsText(Citation c, NaturalNum n)
11      if n = 0 then = ""
12      elseif n < NumAuthorsListed(c)
13          then = Concat(Concat(FirstNAuthorsText(c, Pred(n)),
14                               NameListed(NthMention(c, Pred(n)))),
15                        NthAuthorSeparator(c, Pred(n)))
16      elseif n = NumAuthorsListed(c)
17          then = Concat(Concat(FirstNAuthorsText(c, Pred(n)),
18                                NameListed(NthMention(c, Pred(n)))),
19                        AuthorListTerminator(c));

20  random String AuthorText(Citation c)
21      = FirstNAuthorsText(c, NumAuthorsListed(c));
```

Figure 5.12: The portion of the BLOG model that generates the full author list in a citation.

The next statement defines a distribution for NthAuthorSeparator(c, n), which is the separator between the author mentions at indices n and $n + 1$. The distribution for separator strings is a token unigram model: a model that generates a sequence of tokens independently and puts spaces between them with a certain probability. Here a *token* is a punctuation character or a "word": a maximal sequence of consecutive non-space, non-punctuation characters. This type of model is more than we need to represent author separators, which tend to consist of just one or two tokens (a comma, the word "and", or both — we are ignoring the fact that "and" is more likely to occur in the last separator in an author list). However, we use token unigram models for all the parts of citation strings that are not titles or author names. These models are trained on a set of 515 citations that we parsed and annotated by hand. The various token unigram CPDs in our BLOG model all use the same vocabulary of tokens; token probabilities are smoothed and out-of-vocabulary tokens are handled using the Simple Good-Turing technique [Gale and Sampson, 1995]. We see another token unigram CPD on the next line of Figure 5.12 for AuthorListTerminator(c), the punctuation mark (if any) that marks the end of the author list.

Our model's next task is to describe how the complete author list is assembled from these pieces. This is done using a function symbol FirstNAuthorsText(c, n), whose value is a string containing all author mentions in citation c with indices less than n, along with their trailing separators or terminator. FirstNAuthorsText$(c, 0)$ is an empty string, and the values of FirstNAuthorsText(c, n) for greater values of n are built up recursively. Finally, the function symbol AuthorText(c) gets a value equal to FirstNAuthorsText(c, NumAuthorsListed(c)).

The TitleAsCited function symbol defined in Figure 5.10 yields the title text in a citation, so all that remains is to describe how the complete citation string depends on TitleAsCited and AuthorText. This is done in the last part of our BLOG model, shown in Figure 5.13. First, there is a Boolean function symbol AuthorsBeforeTitle(c)

```
1   random Boolean AuthorsBeforeTitle(Citation c) ~ Bernoulli[0.99];

2   random String InitialFiller(Citation c)
3       ~ TokenUnigram["model/initialFill"];
4   random String MiddleFiller(Citation c)
5       ~ TokenUnigram["model/middleFill"];
6   random String FinalFiller(Citation c)
7       ~ TokenUnigram["model/finalFill"];

8   random String Text(Citation c)
9       if AuthorsBeforeTitle(c)
10          then = Concat(InitialFiller(c),
11                        Concat(AuthorText(c),
12                               Concat(MiddleFiller(c),
13                                      Concat(TitleAsCited(c),
14                                             FinalFiller(c)))))
15      else = Concat(InitialFiller(c),
16                    Concat(TitleAsCited(c),
17                           Concat(MiddleFiller(c),
18                                  Concat(AuthorText(c),
19                                         FinalFiller(c)))));
```

Figure 5.13: The end of the BLOG model for our application.

representing the relative order of the title and author list in citation c. Given this choice, there are three parts of the citation string that need to be filled in: the initial segment before the first author or title segment (often empty, but sometimes containing a citation key such as "[Lashkari et al 94]"); the middle segment between the author and title (often a date); and the final segment after the last title or author segment, which contains the conference or journal name, publisher, etc. The text of each of these "filler" segments has a token unigram distribution. Finally, the text of the citation, denoted by $\mathsf{Text}(c)$, is generated by concatenating these segments in the order determined by $\mathsf{AuthorsBeforeTitle}(c)$.

Figures 5.9–5.13 show the entire BLOG model for our citation matching ap-

plication. This model reproduces the probability distribution used in our earlier hand-coded Java implementation; and unlike the BLOG model we used in Chapter 4, it uses elementary CPDs that are in fact fairly elementary. The parameters of many of these CPDs — specifically, the n-gram models for author names, titles, and other parts of a citation — are learned from separate labeled data (it would also be good to estimate the parameters of the string edit CPD and other parts of our model from data, but we have not implemented this yet).

This BLOG model is admittedly rather long, but we see this as an unavoidable consequence of representing such a detailed model in a general-purpose language. Our BLOG inference engine can load the model and use it to define a prior distribution in our MCMC framework. The evidence we assert includes the value of $\mathsf{Text}(\mathsf{Cit}i)$ for each guaranteed citation $\mathsf{Cit}i$, and we query the truth values of the sentences $\mathsf{PubCited}(\mathsf{Cit}i) = \mathsf{PubCited}(\mathsf{Cit}j)$ for each pair of distinct guaranteed citations $\mathsf{Cit}i$, $\mathsf{Cit}j$.

5.5.3 Proposal distribution

We implemented an application-specific proposal distribution for our citation-matching model. This proposer is based loosely on the one used by Pasula *et al.* [2003], and more directly on our 2003 Java implementation of the MCMC citation matching algorithm. Note that our 2003 implementation was entirely application-specific: all parts of the software, including the data structures for representing MCMC states, the code for computing acceptance probabilities, and the proposal distribution, were hand-coded for the citation-matching task. The proposal distribution that we plug into our MCMC framework reproduces only one component of the 2003 implementation.

Both the entirely hand-coded implementation and our BLOG proposal distribu-

tion use one basic kind of move, namely splitting and merging publications — or to put it more accurately, splitting and merging the clusters of citations that refer to hypothesized publications. When clusters are split or merged, the proposer also proposes new values for the hidden attributes of the affected citations (such as TitleAsCited) and the attributes of the affected publications and researchers. To explore the full space of possible worlds, the proposer should also propose moves that change the total numbers of publications and citations (including ones whose attributes are not instantiated in the current MCMC state), as well as moves that split and merge researchers. However, we did not include such moves in our entirely hand-coded implementation, and we did not add them in our BLOG proposal distribution (this is one way in which our proposer is less sophisticated than that of Pasula *et al.* [2003]). We simply fix the numbers of publications and researchers at 10 times the number of observed citations. As to the researchers, we create separate Researcher objects for all authors of all publications: in our MCMC states, each hypothesized Researcher object serves as the value of $\mathsf{NthAuthor}(p, n)$ for exactly one pair (p, n). These simplifications make our proposer technically incorrect, but do not seem to be the source of most of the errors that we see in the final citation clusterings.

The method we use for proposing split-merge moves is the "simple random split-merge procedure" of Jain and Neal [2004]. We begin by choosing two distinct citations c_1 and c_2. Let p_1 and p_2 be the publications referred to by these citations in the current MCMC state. If $p_1 = p_2$, then we propose to split p_1; otherwise, we propose to merge p_1 and p_2. If we are proposing a split, then we introduce a new object identifier p' for the new publication. We set $V_{\mathsf{PubCited}}[c_1]$ to p_1 and $V_{\mathsf{PubCited}}[c_2]$ to p'. Then for each other citation c that currently refers to p_1, we set $V_{\mathsf{PubCited}}[c]$ to either p_1 or p' with equal probability. For a merge move, we do not have any more choices to make: we just assign all the citations currently referring to p_2 to p_1, so p_2

is left with no citations. Note that if the move we propose is a split, then its reverse move (used to compute the proposal ratio $q(s|s')/q(s'|s)$) is a merge move, and *vice versa*.

A naive implementation of this method would simply choose the two citations c_1 and c_2 uniformly at random. However, such a proposer would often propose merges on citations that have nothing in common, and are clearly not coreferent. Thus, following Pasula *et al.* [2003], we restrict the way c_1 and c_2 are chosen. Specifically, in a preprocessing step, we group the citations into loose, potentially overlapping clusters called *canopies* [McCallum *et al.*, 2000] based on the fraction of words that they have in common. To propose a split-merge move, we first choose a canopy uniformly at random, and then choose two distinct citations uniformly at random from that canopy. This technique vastly increases the proportion of plausible moves among our split-merge proposals.

Taking advantage of the flexibility offered by our MCMC framework, our proposer proposes MCMC states that are abstract partial instantiations of the basic random variables. We use object identifiers to represent publications and researchers. The variables instantiated in each proposed state are the active ancestors of the query and evidence variables. Thus, when a researcher or publication no longer serves as a value for any Referent or PubCited variables, we uninstatiate all variables defined on that researcher or publication. We also uninstantiate variables defined on AuthorMention objects that do not exist in a proposed state, and variables such as FirstNAuthorsText(c, n) where n is not less than the proposed value of NumAuthorsListed(n). Thus, the instantiations that we propose are minimally self-supporting beyond the query and evidence variables, and thus define non-overlapping sets.

Whenever we propose a split or merge move, we also propose changes to the attributes of the affected publications, and the attributes of all citations that refer

to them. Our method for proposing values for these attributes works from the bottom up. The first step is to propose values for the hidden attributes on each citation and its associated author mentions — such as TitleAsCited, SurnameAsCited, and NameReversed — that are consistent with a reasonable segmentation of the citation into fields. To obtain reasonable segmentations, we use a hidden Markov model (HMM) [Rabiner, 1989] in which the outputs are tokens and the hidden states are segment labels such as title, author, authorSeparator, etc. This HMM is trained on the same set of hand-parsed citations that we use to train the token unigram distributions in our BLOG model. There are more recent segmentation techniques that tend to be more accurate than an HMM [Lafferty *et al.*, 2001], but our MCMC process should converge to consistent segmentations for citations that refer to the same publication, so this weak segmentation model should not hurt us too much. Our proposer uses the HMM to compute the 20 best segmentations for each citation. Then whenever it needs to propose new values for a citation's attributes, the proposer samples one of these segmentations (with probabilities proportional to their posterior probabilities in the HMM) and sets the values of random variables on that citation and its associated author mentions accordingly.

To propose attributes for a publication and its authors, our proposer uses a very simple technique: it randomly chooses a citation referring to that publication, and treats this citation's attributes as "authoritative". That is, it proposes values that are equal to the TitleAsCited, SurnameAsCited, and GivenNameAsCited attributes on the citation and its author mentions. This method again fails to explore most of the outcome space, in that it does not propose attribute values that are not seen in the observed citation strings.

The proposal distribution we have described here is not particularly sophisticated, and includes a number of shortcuts that are not technically justified. However, this the same proposal distribution that we used in our entirely hand-coded implemen-

tation in 2003. The main point is that our BLOG MCMC framework supports the same proposal distribution, which proposes abstract partial instantiations. A simpler MCMC framework that required full, concrete instantiations would not support the same proposer.

5.5.4 Experimental results

We evaluated both our BLOG MCMC implementation and our entirely hand-coded Java implementation on four files of citations originally used by Lawrence *et al.* [1999b]. As we described in Section 5.5.1, these files were obtained by searching for particular substrings — specifically "face", "reinforcement", "reasoning", and "constraint" — in a larger collection. The files contain between 295 and 514 citations, which appear as raw text strings without any segmentation. Note that no parts of our BLOG model or proposal distribution were trained on any of these files.

Table 5.1 shows accuracy and speed results for four systems on the four citation files. The accuracy metric here is cluster recall: the percentage of true citation clusters recovered exactly. The first system in the table is a re-implementation of the Lawrence *et al.* [1999a] phrase matching algorithm, written by Pasula *et al.* [2003].[10] This algorithm achieves the lowest accuracies overall. The next system is the Metropolis-Hastings implementation by Pasula *et al.*, which consists of application-specific Lisp code with a sophisticated proposal distribution. This system achieves the best accuracies by a considerable margin (its accuracy is comparable to that of a more recent discriminative approach developed by Wellner *et al.* [2004]). We do not know how many MCMC samples were used to generate the Pasula *et al.* results, although we do know that the accuracy results were averaged over all the

[10]The accuracy results from this re-implementation are used in Pasula *et al.* [2003], although they are considerably lower than the results reported by Lawrence *et al.* [1999a]. This may be because Lawrence *et al.* preprocess the citations with certain normalization operations, which are not fully described in their paper.

	Face 349 citations	Reinforcement 406 citations	Reasoning 514 citations	Constraint 295 citations
Phrase matching				
accuracy	94%	79%	86%	89%
M-H: Pasula *et al.*				
accuracy (avg)	97%	94%	96%	93%
M-H: hand-coded				
accuracy (avg)	95.5%	81.5%	88.6%	91.7%
accuracy (final)	96.1%	86.2%	90.0%	92.0%
time	14.3 s	19.4 s	19.5 s	12.2 s
M-H: BLOG				
accuracy (avg)	95.5%	78.3%	88.7%	90.9%
accuracy (final)	96.4%	82.9%	90.4%	91.5%
time	71.7 s	99.3 s	99.7 s	60.3 s

Table 5.1: Citation matching results for the phrase matching algorithm of [Lawrence *et al.*, 1999], the hand-coded M-H implementation used by [Pasula *et al.*, 2003], a simpler M-H implementation hand-coded in Java, and the BLOG inference engine. Results for the last two systems are averaged over 10 independent runs.

MCMC samples in each run. We also lack data on how long these runs took.

We are primarily interested in comparing the last two systems in the table: our entirely application-specific Java implementation from 2003, and the BLOG MCMC engine using our custom proposal distribution. For these systems, we did 10 separate runs of 10,000 MCMC samples each. We report two versions of the accuracy metric: one averaged over all the MCMC samples (with no burn-in period), and one computed just on the final MCMC state. The two systems achieve similar accuracies, which is to be expected since they implement approximately the same model and proposal distribution. Reflecting the shortcomings of this proposal distribution (and perhaps the model as well), these accuracies are considerably lower than those achieved by Pasula *et al.*

The most interesting aspect of these results is the running times, which tell us how much we sacrifice by using a general-purpose MCMC implementation rather

than one coded entirely for a specific application. The timing results in Table 5.1 reflect the time required to initialize the system and run MCMC for 10,000 samples. Both systems display significant variation in run time across data sets; this reflects differences in the average number of citations affected by split-merge moves (the data sets have different ratios of citations to publications) and differences in the fraction of proposals that are accepted.

Despite this variation, the BLOG engine consistently takes five times as long as the hand-coded Java implementation. One reason why the BLOG engine is slower is that it must store all basic variable values in a general-purpose data structure (a hash table) rather than using Java classes specific to publications and citations. Another reason is that when computing acceptance probabilities, the BLOG engine must determine which factors need to be computed by looking at a context-specific dependency graph, as described in Section 5.4.2.2. The hand-coded implementation contains special code to compute acceptance probabilities for the particular moves that we use. Finally, the compute the probability of a variable given its active parents, the BLOG engine evaluates the terms, formulas, and set expressions in the relevant dependency statement. In the hand-coded version, this evaluation is again implemented by application-specific code.

Still, our notes from the summer of 2003 indicate that the hand-coded implementation ran in about 120 seconds with the computers and Java runtime environment that we had then. In other words, our computing infrastructure has improved enough that a general-purpose system runs as fast as a hand-coded system did three years ago. It is also notable that the speed ratio between the BLOG implementation and the hand-coded one is constant, regardless of variations in the size of the citation file. This suggests that we have been successful in our efforts to make the per-sample computation time independent of the number of instantiated variables, as discussed in Section 5.4.2.2.

Chapter 6

Related Work

The standard formalisms for probabilistic knowledge representation are Bayesian networks (BNs) and their undirected analogues, which explicitly represent a finite set of variables with a fixed dependency structure. There have been many proposals for formalisms that go beyond this baseline by allowing contingent dependencies or infinite sets of variables, or by specifying the model at a first-order rather than propositional level. In this chapter, we review some of these formalisms and discuss their relationship to BLOG.

6.1 Contingent dependencies

There are a number of formalisms for representing context-specific independence (CSI) in BNs. Boutilier et al. [1996] use decision trees, just as we do in CBNs. Poole and Zhang [2003] use a set of *parent contexts* (partial instantiations of the parents) for each node; such models can be represented as PBMs, although not necessarily as CBNs. Neither paper discusses infinite or cyclic models. The idea of labeling edges with the conditions under which they are active may have originated with Fung and Shachter [1990] (a working paper that is no longer available); it was recently revived

by Heckerman *et al.* [2004].

Bayesian multinets [Geiger and Heckerman, 1996] can represent models that would be cyclic if they were drawn as ordinary BNs. A multinet is a mixture of BNs: to sample an outcome from a multinet, one first samples a value for the *hypothesis variable* H, and then samples the remaining variables using a hypothesis-specific BN. We could extend this approach to CBNs, representing a structurally well-defined CBN as a (possibly infinite) mixture of acyclic, finite-ancestor-set BNs. However, the number of hypothesis-specific BNs required would often be exponential in the number of variables that govern the dependency structure. On the other hand, to represent a given multinet as a CBN, we simply convert each edge $X \to Y$ in the hypothesis-specific BN for hypothesis h into a CBN edge $X \to Y$ with the label $H = h$.

6.2 Infinite models

As we mentioned at the beginning of Section 2.4.2, several other authors have discussed conditions under which a BN with infinitely many nodes defines a unique distribution. Pfeffer [2000] and Kersting and de Raedt [2001a] give essentially the same results we gave for non-contingent BNs in Section 2.4.2, although their theorems make reference to particular first-order languages for describing such BNs.

We are not aware of any previous work that exploits the contingent nature of dependencies to obtain stronger results on when a model defines a unique distribution. However, there are some other results that go beyond ours in the non-contingent case. Jaeger [1998] states that an infinite BN defines a unique distribution if there is a well-founded topological ordering (not necessarily a numbering) on its variables: that condition is more complete than the ones we give for CBNs in that it allows a node to have infinitely many active parents, but less complete in that it requires

a single ordering for all contexts. Laskey [2006] shows that an infinite BN is well-defined if its nodes can be grouped into a sequence of numbered levels, such that each edge into a level-d node comes from a node at level $d-1$ or earlier. This criterion allows infinite parent sets, but is weaker than Jaeger's well-founded ordering criterion in that it does not allow a node X to have parents at an infinite number of levels (then there would be no subsequent level for X to be in). Pfeffer and Koller [2000] point out that a network containing an infinite receding path $X_1 \leftarrow X_2 \leftarrow X_3 \leftarrow \cdots$ may still define a unique distribution if the CPDs along the path form a Markov chain with a unique stationary distribution.

Another line of work on infinite models deals with the case of undirected graphical models, also called *Markov networks* or *Markov random fields*. Most work on Markov networks in AI deals with finite networks [Pearl, 1988]; for the infinite case, see Georgii [1988]. In a Markov network, the joint distribution is specified not with conditional probabilities, but with local *potential functions* ψ_i that assign weights to the instantiations of certain sets of random variables S_i. The set of random variables over which a potential function is defined must be a *clique* in the network: that is, there must be an edge between each pair of variables in the set. In the finite case, the probability of a complete instantiation σ is given by:

$$P(\sigma) = \frac{1}{Z} \prod_i \psi_i(\sigma[S_i])$$

where Z is a normalizing constant that makes the distribution sum to one.

When there are infinitely many variables, the Markov network asserts that for each finite subset S of the variables, the conditional distribution on S given any instantiation of its boundary (*i.e.*, those neighbors of variables in S that are not themselves S) is given by the equation above. A *Gibbs measure* for the network is a probability measure for all the variables that satisfies these constraints. Thus, to

find an analogue to our results about infinite BNs and PBMs, we should ask under what conditions an infinite Markov network has a unique Gibbs measure. It turns out that the uniqueness of the Gibbs measure depends on the parameters of the potential functions: as the parameters are changed, the network can go through a *phase transition* where it switches from one Gibbs measure to another. These results contrast with our results for directed models, where the uniqueness conditions are purely structural and make no mention of the parameters of local distributions.

6.3 First-order probabilistic languages

First-order probabilistic languages (FOPLs) differ from propositional representations such as Bayesian networks in the same way first-order logic differs from propositional logic: they allow the modeler to make statements about all the objects in some class, rather than modeling each object individually. A large number of FOPLs have been proposed by various authors over the last 15 years or so, yielding a sometimes bewildering array of languages. In this section, we discuss some of the major dimensions along which these languages differ, and note now they compare with BLOG. Our discussion yields a classification scheme for FOPL models, shown in Figure 6.1.

Our comparisons of various FOPLs will use the following pedagogical example. Because many FOPLs do not have features for representing unknown objects or contingent dependencies, we use an example where the objects and dependency structure are known.

Example 6.1. *Suppose we are given a list of papers that have been submitted to a conference over several years. Each paper is either accepted or not accepted. We are also given a list of researchers, which includes the primary author of each paper. Suppose that each researcher can be classified as brilliant or not brilliant, and the probability that a paper is accepted depends on whether its primary author is brilliant*

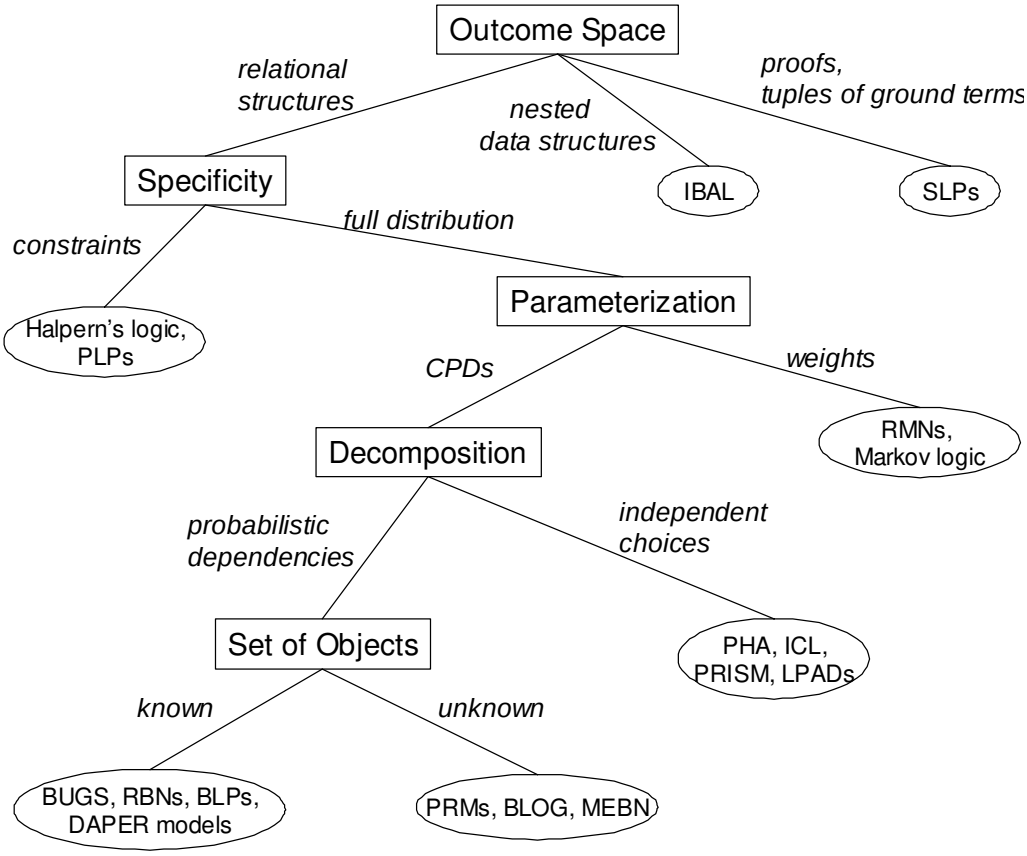


Figure 6.1: A classification scheme for first-order probabilistic languages.

or not. Given the authorship and acceptance status of certain papers, we would like to predict which other papers will be accepted.

6.3.1 Outcome spaces

The most basic way in which certain FOPLs differ from others is in their outcome spaces. In most FOPLs, the outcome space is a set of *relational structures*, which specify a set of objects and some relations (or functions) on these objects. For instance, a relational structure for Example 6.1 specifies a set of papers, a set of researchers, a unary predicate called Accepted on papers, a unary predicate called

Brilliant on researchers, and a function called PrimaryAuthor that maps papers to researchers. Depending on the what aspects of the scenario are known in advance, the outcomes may share some *relational skeleton* [Friedman *et al.*, 1999]: for instance, they may all have the same sets of objects and the same PrimaryAuthor function.

One reason for the diversity of FOPLs is that different communities talk about relational structures in different ways. In this thesis we have taken an approach based on first-order logic, where the relational structures are logical model structures. The idea of defining probability distributions over model structures dates back at least to a paper by Gaifman [1964]. Gaifman shows that if one specifies the probabilities of the events $\{\omega : \omega \models_\alpha \varphi\}$ for all *quantifier-free* formulas φ and all assignments α to the free variables in φ, then the satisfaction probabilities for sentences with quantifiers are uniquely determined. Gaifman does not actually define a modeling language, *i.e.*, a way of specifying all the necessary probabilities in a finite amount of space.

Since Gaifman's work, however, researchers have proposed many FOPLs that define distributions over logical model structures. Examples include Halpern's logic of probability on possible worlds [Halpern, 1990], probabilistic Horn abduction (PHA) [Poole, 1993], relational Bayesian networks (RBNs) [Jaeger, 1997], PRISM [Sato and Kameya, 1997], Markov logic [Richardson and Domingos, 2006] and multi-entity Bayesian networks (MEBN) [Laskey, 2006]. In some other areas of computer science, relational structures are thought of as instances of a relational database schema. This view has led to a distinct set of FOPLs, including probabilistic relational models (PRMs) [Koller and Pfeffer, 1998; Friedman *et al.*, 1999] and relational Markov networks [Taskar *et al.*, 2002].

The statistics community thinks of possible outcomes in yet another way: as instantiations of a set of random variables. The statistical analogue of the unary predicate Accepted is a family of binary-valued random variables A_i, indexed by nat-

ural numbers i that represent papers. Similarly, the function PrimaryAuthor can be represented as an indexed family of random variables P_i, whose values are natural numbers representing researchers. Thus, the instantiations of a set of random variables can represent relational structures (at least when the set of objects is fixed). Indexed families of random variables can be represented graphically using "plates" that contain co-indexed nodes; they also form the foundation for the modeling language used by the BUGS system [Thomas *et al.*, 1992].

Bayesian logic programs (BLPs) [Kersting and De Raedt, 2001a] also fall into this variable-based category, although in a somewhat counterintuitive way. A BLP uses a logic program to define a Bayesian network: the variables in the BN correspond to ground atoms (atomic formulas) that are provable in the logic program. Whereas logical atoms take on Boolean values by definition, the variable corresponding to a ground atom such as Height(John) may be real-valued. Thus, while the BN defined by a BLP can be thought of as defining a distribution over relational structures, these are not model structures for the logical language used in the program.

There are two well-known FOPLs whose possible outcomes are not relational structures in the sense we have defined. One is stochastic logic programs (SLPs) [Muggleton, 1996]. An SLP defines a distribution over proofs from a given logic program. If a particular goal predicate R is specified, then an SLP also defines a distribution over tuples of logical terms: the probability of a tuple $(t_1, \ldots, t_k)$ is the sum of the probabilities of proofs of $R(t_1, \ldots, t_k)$. SLPs are useful for defining distributions over objects that can be encoded as terms, such as strings or trees; they can also emulate more standard FOPLs [Puech and Muggleton, 2003].

The other prominent FOPL with a unique outcome space is IBAL [Pfeffer, 2001], a programming language that allows stochastic choices. An IBAL program defines a distribution over *environments* that map symbols to values. These values may be individual symbols, like the values of variables in a BN; but they may also be other

environments, or even functions. An IBAL program could implement a BLOG-like generative process if the output values were interpreted as logical model structures. But since this interpretation would not be explicit in the language, the declarative semantics of such a program would be less clear than the corresponding BLOG model.

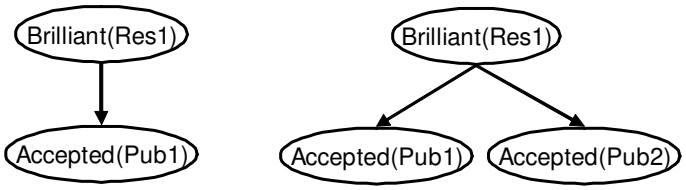
6.3.2 Specificity

Among the FOPLs that define distributions over relational structures, the first distinction we can draw is between languages that fully define a distribution, and those that only impose constraints on a distribution. As an example of the latter type, Halpern's logic of probability on possible worlds [Halpern, 1990] allows statements such as $\forall x\, P(\mathsf{Brilliant}(x)) = 0.3$. Such statements just specify particular marginal probabilities: in general, they do not fully define a distribution. Probabilistic logic programs (PLPs) [Ng and Subrahmanian, 1992] are essentially a version of Halpern's language restricted to Horn clauses, although one can obtain a full distribution from a PLP by finding the maximum entropy distribution consistent with the PLP's constraints [Lukasiewicz and Kern-Isberner, 1999].

In BLOG, on the other hand, we consider a model well-defined only if it fully defines a distribution over relational structures. Most of the other FOPLs we have mentioned also fall into this category; we will focus on them from here on.

6.3.3 Conditional probabilities versus weights

In the propositional realm, Bayesian networks are directed models that specify a conditional probability distribution (CPD) for each variable given some parent variables, whereas *Markov networks* are undirected models that use weights to define the relative probabilities of instantiations. This distinction carries over to the first-order



(a) (b)

Figure 6.2: BNs defined by a directed FOPL model whose relational skeleton includes (a) one paper, or (b) two papers.

case. Besides BLOG, the CPD-based or directed FOPLs include BUGS [Thomas *et al.*, 1992], PRISM [Sato and Kameya, 1997], PRMs [Koller and Pfeffer, 1998], BLPs [Kersting and De Raedt, 2001a], and MEBN [Laskey, 2006]. The principal weight-based or undirected formalisms are relational Markov networks [Taskar *et al.*, 2002] and Markov logic [Richardson and Domingos, 2006].

To understand the trade-offs between directed and undirected representations, consider a directed FOPL model for Example 6.1 with the following CPDs:

$\mathsf{Brilliant}(r) \sim$

True	False
0.2	0.8

$\mathsf{Accepted}(p) \sim$

	Accepted(p)	
Brilliant(PrimaryAuthor(p))	True	False
True	0.8	0.2
False	0.3	0.7

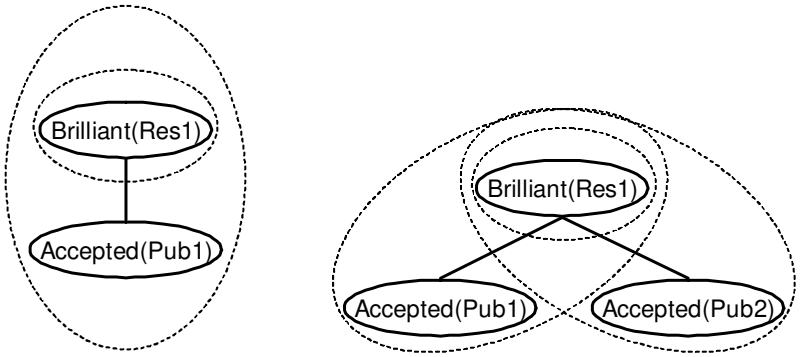
If the relational skeleton contains just one paper Pub1 and just one researcher Res1, with PrimaryAuthor(Pub1) = Res1, then this model defines the BN in Figure 6.2(a). If there are two papers by Res1, we get the BN in Figure 6.2(b).

This directed model has several attractive properties. First, the parameters have clear interpretations as prior and conditional probabilities, and can be estimated from fully observed data using elementary formulas. Even more importantly, the parameters are *modular*: they reflect causal processes that apply regardless of the relational skeleton. Thus, if we estimate the parameters using only examples with one paper per researcher, we will get the same CPDs that we would get from examples with two papers per researcher. We can also exploit a related modularity property when performing inference: rather than doing inference on the whole BN defined by the FOPL model, it suffices to use the subgraph consisting of the query and evidence nodes and their ancestors [Ngo and Haddawy, 1997].

The drawback of directed models is that they must not have any cycles. This requirement is especially burdensome in FOPLs, because we must ensure that the probability model is acyclic for every relational skeleton in some class. Also, certain properties of relations are difficult to describe in a directed model: for instance, we cannot easily specify that a relation on a set of interchangeable objects is symmetric.

Undirected models, on the other hand, have no acyclicity constraints. As we mentioned in Section 6.2, an undirected model is defined by *potential functions* that assign weights to instantiations based on some subsets of the random variables. The weight of an instantiation is the product of the weights assigned by all the potentials; these weights are then normalized to yield a probability distribution. In the first-order case, a model specifies *potential function templates* that apply to all sets of variables that satisfy certain conditions. For instance, in Example 6.1, we can include a potential template that applies to $\mathsf{Brilliant}(r)$ for every researcher r, and another template that applies to $\{\mathsf{Brilliant}(r), \mathsf{Accepted}(p)\}$ for all pairs (r, p) such that $\mathsf{PrimaryAuthor}(p) = r$. Figure 6.3 shows the Markov networks that result when these templates are applied to relational skeletons with one or two papers.

This undirected FOPL model can reproduce the distributions defined by our



(a) (b)

Figure 6.3: Markov networks defined by an undirected FOPL model whose relational skeleton includes (a) one paper, or (b) two papers. Dotted ovals indicate sets of variables that are in the domain of the same potential function.

directed model above: we can simply set the potential on $\mathsf{Brilliant}(r)$ equal to the CPD for $\mathsf{Brilliant}(r)$, and the potential on $\{\mathsf{Brilliant}(r), \mathsf{Accepted}(p)\}$ to the CPD for $\mathsf{Accepted}(p)$. However, suppose we estimate our parameters solely on examples with one paper per researcher (recall that this caused no problems in the directed case). For the network in Figure 6.3(a), the following parameterization assigns the same weight to each full instantiation as the CPD-like one does:

$\forall\, r$:

$\mathsf{Brilliant}(r)$	
True	False
1	1

$\forall\, (r, p)$ s.t. $\mathsf{PrimaryAuthor}(p) = r$:

	$\mathsf{Accepted}(p)$	
$\mathsf{Brilliant}(r)$	True	False
True	0.16	0.04
False	0.24	0.56

A learning algorithm has no reason to prefer the CPD-like parameterization to this one, because both yield the same probability distribution. However, the meanings of the parameters are no longer so clear. For instance, although the potential on $\mathsf{Brilliant}(r)$ is all 1's, $\mathsf{Brilliant}(\mathsf{Res1})$ is still more likely to be True than False in Figure 6.3(a) because the event $\mathsf{Brilliant}(\mathsf{Res1}) = \mathsf{False}$ receives greater total weight in the potential over $\{\mathsf{Brilliant}(\mathsf{Res1}), \mathsf{Accepted}(\mathsf{Pub1})\}$. Because of this coupling between potentials, maximum-likelihood parameters for Markov networks cannot be found with simple formulas: one must use a gradient-based optimization algorithm [Richardson and Domingos, 2006].

Now consider what happens if we apply the undirected probability model above to the two-paper network in Figure 6.3(b). Then the template for pairs (r, p) such that $\mathsf{PrimaryAuthor}(p) = r$ applies twice, and the marginal distribution on $\mathsf{Brilliant}(\mathsf{Res1})$ ends up being proportional to $(0.2^2, 0.8^2)$. If the actual probability that a researcher is brilliant is 0.2, then these parameters are sub-optimal: we would not learn them if we had instances with two papers in our training set.[1] Thus, unlike in the directed case, we need to ensure that the relational skeletons in our training set reflect the diversity of relational skeletons that we may encounter in test data.

6.3.4 Independent choices versus probabilistic dependencies

The category of CPD-based languages for defining complete distributions over relational structures is still quite large. However, two of the languages we have mentioned, namely PHA [Poole, 1993] and PRISM [Sato and Kameya, 1997], stand out from the rest in that they represent only deterministic dependencies and independent random choices. That is, each variable either has no parents, or has a deterministic

[1] The problem actually gets worse if we eliminate the apparently redundant potential template on $\mathsf{Brilliant}(r)$: then there is no parameterization that yields the desired distribution for all relational skeletons.

CPD. Other FOPLs that take this approach include independent choice logic (ICL) [Poole, 1997] and logic programs with annotated disjunctions (LPADs) [Vennekens *et al.*, 2004].

It may not be immediately obvious how independent choices could suffice to represent all the randomness in a probabilistic model. First, consider the directed model that we defined in the previous section for Example 6.1. To sample a value for an $\mathsf{Accepted}(p)$ variable in that model, we flip a coin with a bias determined by the value of $\mathsf{Brilliant}(\mathsf{PrimaryAuthor}(p))$. The trick used in PHA and PRISM is, conceptually, to flip coins for all possible values of $\mathsf{Brilliant}(\mathsf{PrimaryAuthor}(p))$ ahead of time, and then choose which coin flip to use based on the actual value of $\mathsf{Brilliant}(\mathsf{PrimaryAuthor}(p))$. The initial coin flips can be represented by an auxiliary predicate $\mathsf{Accepted\_given\_Brilliant}(p, b)$, which represents the value that $\mathsf{Accepted}(p)$ would have if $\mathsf{Brilliant}(\mathsf{PrimaryAuthor}(p))$ were equal to b. The probability model for $\mathsf{Accepted\_given\_Brilliant}$ is as follows:

$\mathsf{Accepted\_given\_Brilliant}(p, \mathsf{True}) \sim$

True	False
0.8	0.2

$\mathsf{Accepted\_given\_Brilliant}(p, \mathsf{False}) \sim$

True	False
0.3	0.7

Now the probability model for $\mathsf{Accepted}$ is deterministic (note that we are treating $\mathsf{Brilliant}$ here as a Boolean function, yielding values in $\{\mathsf{True}, \mathsf{False}\}$):

$$\mathsf{Accepted}(p) = \mathsf{Accepted\_given\_Brilliant}(p, \mathsf{Brilliant}(\mathsf{PrimaryAuthor}(p)))$$

The advantage of this technique is that it completely separates the logical and probabilistic parts of the language. This separation can be exploited to obtain effi-

cient algorithms for certain tasks [Sato and Kameya, 2001]. However, this decomposition often makes the representation considerably less intuitive.

6.3.5 Known versus unknown objects

The last distinction in our classification scheme is whether a language requires the set of objects to be specified in the relational skeleton, or allows the set of objects to be unknown. Most FOPLs assume that the objects are in one-to-one correspondence with a given set of symbols. Some logic-based FOPLs assume that the objects correspond one-to-one with the ground terms of the language; thus, they define a distribution just over *Herbrand models*. The CPD-based FOPLs that make such assumptions include BUGS [Thomas *et al.*, 1992] (where the objects correspond to specified sets of natural numbers), RBNs [Jaeger, 1997], BLPs [Kersting and De Raedt, 2001a], and directed acyclic probabilistic entity-relationship (DAPER) models [Heckerman *et al.*, 2004]. One can model unknown objects to some extent in these languages by adding an Exists predicate, but one still has to specify all the objects that could exist, and craft the probability models so that objects for which Exists is false cannot serve as values for functions or have any probabilistic influence on other objects.

The need to handle unknown objects has been appreciated since the early days of FOPL research: Charniak and Goldman's *plan recognition networks* [1993] can contain unbounded numbers of objects representing hypothesized plans. However, external rules are used to decide what objects and variables to include in a network. While each possible plan recognition network defines a distribution on its own, Charniak and Goldman do not claim that the various networks constructed by their system are all approximations to some single distribution over outcomes.

Some more recent FOPLs directly define a distribution over outcomes with vary-

ing objects. PRMs and their extensions include various constructs for modeling different kinds of uncertainty about object existence. An early PRM paper [Koller and Pfeffer, 1998] allows *number uncertainty*: uncertainty about the number of objects that stand in a given relation to a single parent object, such as the number of passengers in a car. Getoor *et al.* [2001] introduce *existence uncertainty*: uncertainty about whether there exists an object that stands in a certain relation to two or more given objects, such as a role for a given actor in a given movie. Finally, Pasula *et al.* [2003] augment PRMs with what could be called *domain uncertainty*: uncertainty about the total number of objects of some class, such as the number of publications in a field. However, there is no unified syntax or semantics for dealing with unknown objects in PRMs. MEBNs [Laskey, 2006] take yet another approach: an MEBN model includes a set of unique identifiers, and there is a random variable for each identifier indicating whether the object it denotes exists or not. However, the modeler must still specify the list of objects that may exist.

Our approach to unknown objects in BLOG can be seen as unifying the PRM and MEBN approaches. Number statements neatly generalize the various ways of handling unknown objects in PRMs: number uncertainty corresponds to a number statement with a single origin function; existence uncertainty can be modeled with two or more origin functions (and a CPD whose support is $\{0, 1\}$); and domain uncertainty corresponds to a number statement with no origin functions. There is also a correspondence between BLOG and MEBN logic: the tuple representations in a BLOG model can be thought of as unique identifiers in an MEBN model. The difference is that BLOG determines which objects actually exist in a world using number variables rather than individual existence variables.

There is also a relatively new FOPL called *dynamical grammars* [Mjolsness, 2006] that can be seen as going even farther than BLOG in the direction of unknown objects. The basic element in dynamical grammars is a *parameterized term*, which

represents an object and all its properties. For instance, the term $\mathsf{bacterium}(x)$ represents a bacterium whose position is a vector x. A dynamical grammar specifies a probabilistic model of how a multiset of terms evolves over time. The grammar consists of rules that replace a set of input terms with a set of output terms. Each rule has a "firing rate" expression that defines the probability of the rule being executed on any given set of input terms per infinitessimal unit of time. For example, a rule could specify that a bacterium and an immune system cell are replaced with just an immune system cell (the bacterium is engulfed) with some firing rate that depends on the position vectors in the two input terms. This language is well suited for describing how objects are created and destroyed over time, but it also has some awkward features: for instance, the term representation of an object must have an argument for each attribute of that object that is relevant in any part of the model.

Chapter 7

Conclusion

7.1 Contributions of this thesis

This thesis has introduced a new framework for probabilistic reasoning about unknown objects, ranging from the unknown aircraft that generated a set of radar blips to the unknown publications that underlie a set of bibliographic citations. The centerpiece of the thesis is Bayesian logic (BLOG), a representation language that concisely defines probability distributions over possible worlds with varying sets of objects and varying relations among the objects.

Representing probability distributions over such complex outcome spaces has required us to address a number of technical challenges. One difficulty is that when the relationships between objects vary from outcome to outcome, the probabilistic dependencies between random variables often vary as well. That is, the dependencies become contingent rather than fixed. For example, in our hurricane scenario (Example 3.2), the dependencies between damage levels and preparations in different cities are contingent on which city is hit first. Another difficulty is that if the number of objects in the world is unknown and unbounded, then the number of variables in the

model often must be infinite. For instance, in the urn-and-balls example (Example 3.1), we cannot limit ourselves to any finite number of TrueColor variables, because then we could not represent outcomes where the number of balls exceeded that upper bound. A third issue is that when the set of objects is unknown, it may no longer be appropriate for the outcome space to be the product space of a fixed set of random variables. For instance, a variable representing the true color of ball 37 should not be allowed a full range of values in worlds where the number of balls is only 12.

We addressed all these challenges in Chapter 3, where we defined two new declarative modeling formalisms: partition-based models (PBMs) and contingent Bayesian networks (CBNs). Both of these formalisms allow infinite sets of variables, as well as non-product outcome spaces in which some instantiations of the random variables may be unachievable. PBMs are very abstract and general: they allow one to specify conditional probabilities for a random variable not just given some parent variables, but given an arbitrary partition of the outcome space. In Section 3.4, we proved that the probability distribution defined by a PBM can be characterized either by certain context-specific independence properties, or equivalently by factorization properties for certain finite instantiations of the random variables (Theorem 3.10). We also gave a general condition under which a PBM is guaranteed to define a unique distribution (Theorem 3.13).

CBNs are more down-to-earth, using decision trees to define partitions, and representing contingent dependencies by labeling graph edges with the conditions under which they are active. Using decision trees to define the partitions deprives CBNs of some expressive power, but also facilitates the development of inference algorithms (such as the likelihood weighting algorithm in Section 5.3). The graphical representation also allows us to define graph-based criteria for checking that a CBN defines a distribution (Theorem 3.17). These criteria can be satisfied even when the CBN contains cycles, or when some nodes in the CBN have infinitely many parents —

as long as the edges involved in these cycles or infinite parent sets are labeled with contradictory conditions.

In Chapter 4, we introduced the BLOG language itself. BLOG is intended to be a convenient modeling language for scenarios with unknown objects, as well as for a full range of other probabilistic knowledge representation tasks. Intuitively, a BLOG model can be thought of as defining a generative process that constructs a possible world step by step. In addition to setting the values of functions on tuples of objects, these steps can add varying numbers of new objects to the world. One of the distinctive features of BLOG is that it lets a modeler define processes in which objects generate other objects — for example, aircraft generate radar blips — and the number of objects generated can depend on various attributes of the generating objects.

More formally, a BLOG model is a concise description of a PBM, specifying conditional probability distributions and context-specific independence properties. In general, defining a PBM for a scenario with unknown objects can be tricky: one needs to make sure that the random variables in the PBM resolve all questions about the outcomes (*e.g.*, which unknown objects end up playing which roles in a relational structure), and also that unachievable instantiations of the random variables (instantiations not corresponding to any outcome) get probability zero. In Section 4.3, we described a semantics for BLOG that frees the modeler from the need to worry about such issues: it is impossible to write a BLOG model that creates a mismatch between the random variables and the outcome space (see Lemma 4.2 and Proposition 4.6). Then in Section 4.3.6, we built on our results from Chapter 3 to define a class of *structurally well-defined* BLOG models, which are guaranteed to define probability distributions.

Chapter 5 discussed the task of doing inference in BLOG models. We presented variants of three standard sampling-based inference techniques: rejection sampling,

likelihood weighting, and Markov chain Monte Carlo (MCMC). In all three cases, we extended the standard algorithms to avoid constructing complete instantiations of the set of random variables, which is often infinite. Instead, they operate on partial descriptions of possible worlds. The rejection sampling and likelihood weighting algorithms are guaranteed to converge to correct posterior probabilities for all queries on structurally well-defined BLOG models. However, these guarantees are only asymptotic: in practice, these algorithms often fail to yield useful results even with hours of computation time.

We were able to do some experiments on our urn-and-balls model using the likelihood weighting algorithm. For a real test, however, we turned to our MCMC framework. This framework (Section 5.4) allows a programmer to plug in an arbitrary proposal distribution, which can guide the Markov chain toward worlds with high posterior probability. The proposals may be partial world descriptions that only instantiate context-specifically relevant variables; they may even abstract away the identities of interchangeable objects. As described in Section 5.5, we evaluated this framework on a real-world task that involves clustering bibliographic citation strings into groups that refer to the same publication. Using a customized proposal distribution, we were able to generate 10,000 samples and achieve reasonable accuracy in just 1 or 2 minutes. This was not as fast as an MCMC program written by hand for a particular model and proposal distribution, but it is still an encouraging result.

Our inference engine is implemented in Java, and is publicly available from the author's home page. The engine can read in any BLOG model from a file and compute posterior probabilities using either rejection sampling or likelihood weighting. It can also run our MCMC algorithm using either a very weak default proposal distribution, or a special-purpose proposer implemented by the user as a Java class.

7.2 Directions for future research

This thesis opens up a number of interesting avenues for future research. One important shortcoming of the current work is that PBMs and CBNs are limited to discrete random variables. Much of the development of PBMs revolves around instantiations of the random variables — which would all have probability zero if the variables were continuous. It seems that it should be possible to generalize PBMs and CBNs to deal with probability *densities* on the random variables, but it is not yet clear how to do this. Another goal is to allow variables in CBNs to have infinitely many active parents. This would require moving from numberings of the variables to well-founded orderings (as suggested by Jaeger [1998]), and using transfinite induction. However, this extension raises some of the same difficulties as allowing continuous random variables, since an instantiation of an infinite set of discrete random variables also has probability zero in most cases.

There are also some ways in which we could improve the syntax and expressiveness of BLOG. For instance, it would be convenient to treat sets and lists as first-class objects. A major but probably desirable change would be to make BLOG more compositional — so that, for example, one BLOG model could be used as an elementary CPD within a larger BLOG model. This would require some BLOG models to define conditional distributions, rather than just prior distributions; it is not yet obvious how the semantics would work.

In terms of inference for BLOG models, we are still quite far from our goal of letting a user write down a new model for a complex scenario, provide some observed data, press "Go", and get accurate results from an inference engine within a reasonable amount of time. Currently, achieving efficient inference on a complex model requires using our MCMC framework with a customized proposal distribution. And implementing a proposal distribution is difficult: the programmer must

handle all the details of determining which variables need to be instantiated in a proposed state, computing the forward and backward proposal probabilities, and so on. We would like to develop an architecture that allows proposal distributions to be assembled from components. Then rather than implementing a domain-specific proposal distribution from scratch, a user could choose appropriate components from a library, or write new components for particular parts of a model. It might also help to allow certain parameters in a proposal distribution to be adapted as the chain is running, as in the work of Haario *et al.* [2001]. Another obvious improvement to our inference engine's capabilities would be to include an implementation of Gibbs sampling, as in the BUGS system [Thomas *et al.*, 1992]. There is some research to be done on extending Gibbs sampling to deal with partial instantiations, as we did with likelihood weighting and general Metropolis-Hastings algorithms.

Although we have focused on sampling-based approaches to inference in this thesis, the declarative semantics of BLOG yields inference problems that could be tackled with any class of algorithms. Implementing a variety of algorithms in our BLOG engine would allow users more flexibility in choosing the right algorithm for a particular model. Among approximate inference algorithms, the most promising candidates for implementation are loopy belief propagation [Murphy *et al.*, 1999] and a structured mean-field algorithm [Xing *et al.*, 2003]. Extending these algorithms to handle infinite models with contingent dependencies is an interesting research topic. We would also like to investigate the application of exact inference algorithms to BLOG models — either on their own, or in combination with an MCMC method that samples values for some of the random variables and allows the exact inference algorithm to sum out the rest (a strategy called Rao-Blackwellisation [Casella and Robert, 1996]). There is a large amount of intriguing work these days on exact inference algorithms that exploit determinism and context-specific dependencies, using techniques such as compilation to an arithmetic circuit [Chavira *et al.*, 2006]

or weighted model counting on propositional formulas [Sang *et al.*, 2005]. BLOG models reveal a great deal of structure that such algorithms could exploit. Lifted inference algorithms that exploit generalization across objects [Pfeffer *et al.*, 1999; Poole, 2003; de Salvo Braz *et al.*, 2005] also apply naturally to BLOG, and might yield dramatic speed-ups on some problems.

There is also a set of questions regarding *dynamic* BLOG models: models that describe the evolution of a set of objects over time. BLOG can already represent such models — the aircraft-tracking model in Figure 4.4 is one of them — but our inference algorithms run on a single batch of observed data, rather than updating state estimates incrementally over time. We would like to develop *filtering* algorithms that allow a system to compute posterior probabilities for random variables at the current time step, based on the probabilities computed at the previous time step and any new observations that were received. Ideally, such filtering algorithms should require a constant amount of computation per time step. But in models where one can observe different sets of objects over time, and objects that are no longer observed may still reappear or have some effect on the query variables, it is not clear if constant-time filtering is possible. A system that always reasons about the current states of all the objects it has ever observed will get slower and slower over time. Thus, one research goal is to develop approximate filtering algorithms that avoid such slowdowns.

Learning BLOG models is also a major topic for future research. Parameter estimation from complete data is straightforward; for incomplete data, we plan to add a Monte Carlo version of the expectation maximization (EM) algorithm [Wei and Tanner, 1990] to our MCMC engine. Learning the dependency structure of BLOG models is a more difficult issue. There are known algorithms for learning the structure of probabilistic relational models (PRMs) [Friedman *et al.*, 1999], but the dependencies between variables in a PRM are represented just with *slot chains*,

which are much less expressive than BLOG's dependency statements. Thinking even more ambitiously, we would like to go beyond learning the dependency structure of a model with known object types and function symbols: we would like to have the learner invent new types and functions to explain the observed data. This idea has been studied in the inductive logic programming community under the heading of *predicate invention* [Muggleton and Buntine, 1988]. Recently there has been some work on extending predicate invention techniques to probabilistic models [Otero and Muggleton, 2006; Revoredo *et al.*, 2006]; in a probabilistic setting, inventing a new random predicate corresponds to inventing an indexed family of hidden variables [Elidan and Friedman, 2005].

Finally, we would like to explore further applications of BLOG to real-world problems. The citation-matching model and proposal distribution that we described in Section 5.5 do not yield accuracies as high as state-of-the-art systems [Pasula *et al.*, 2003; Wellner *et al.*, 2004]. Also, identifying citations that refer to the same publication is only part of the task of constructing a bibliographic database: we would also like to identify the distinct authors, journals, conferences, and publishers that are mentioned in a set of citations. There has been work on such multi-type coreference resolution in other probabilistic frameworks [Culotta and McCallum, 2005; Singla and Domingos, 2006]. Representing multiple types of objects BLOG is straightforward, but implementing a good proposal distribution for such an extended model becomes very complex. Applying MCMC to such complex models will be much easier if we develop a component-based architecture for proposal distributions.

There are many other real-world tasks where the BLOG approach should be useful. Examples include: determining when several noun phrases refer to the same object, either within a single document [Soon *et al.*, 2001] or across documents [Li *et al.*, 2004]; identifying multiple web pages that offer the same product in an online shopping system; reconstructing phylogenetic trees for sets of species based on their

genomes [Felsenstein, 2003]; or keeping track of the people and objects encountered by a mobile robot. Tackling such a wide range of tasks will require improved general-purpose algorithms for inference and learning in BLOG. In the longer term, a BLOG-like language could serve as the foundation for general-purpose question-answering systems, personal assistants, or autonomous robots. Such systems would need to learn models for many different domains, from transportation to basic medicine to cooking. Perhaps the most exciting aspect of BLOG is that it gives us an idea of how such general-purpose systems might represent and reason about the world.

Bibliography

[Andrieu *et al.*, 2003] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50:5–43, 2003.

[Bahl and Jelinek, 1975] L. R. Bahl and F. Jelinek. Decoding for channels with insertions, deletions, and substitutions with applications to speech recognition. *IEEE Trans. Inform. Theory*, 21(4):404–411, 1975.

[Bahl *et al.*, 1983] L. R. Bahl, F. Jelinek, and R. L. Mercer. A maximum likelihood approach to continuous speech recognition. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 5(2):179–190, 1983.

[Bar-Shalom and Fortmann, 1988] Y. Bar-Shalom and T. E. Fortmann. *Tracking and Data Association*. Academic Press, Boston, 1988.

[Billingsley, 1995] P. Billingsley. *Probability and Measure*. Wiley, New York, 3rd edition, 1995.

[Borchers *et al.*, 2002] D. L. Borchers, S. T. Buckland, and W. Zucchini. *Estimating Animal Abundance*. Springer, New York, 2002.

[Boutilier *et al.*, 1996] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *Proc. 12th Conf. on Uncertainty in Artificial Intelligence*, pages 115–123, 1996.

[Casella and Robert, 1996] G. Casella and C. P. Robert. Rao-Blackwellisation of sampling schemes. *Biometrika*, 83(1):81–94, 1996.

[Charniak and Goldman, 1993] E. Charniak and R. P. Goldman. A Bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[Chavira *et al.*, 2006] M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational Bayesian networks for exact inference. *Int'l J. Approximate Reasoning*, 42:4–20, 2006.

[Cheeseman *et al.*, 1988] P. Cheeseman, J. Kelly, M. Self, J. Stutz, W. Taylor, and D. Freeman. AutoClass: A Bayesian classification system. In *Proc. 5th Int'l Conf. on Machine Learning*, pages 54–64, 1988.

[Chickering *et al.*, 1997] D.M. Chickering, D. Heckerman, and C. Meek. A Bayesian approach to learning Bayesian networks with local structure. In *Proc. 13th Conf. on Uncertainty in Artificial Intelligence*, pages 80–89, 1997.

[Church, 1936] A. Church. A note on the Entscheidungsproblem. *J. Symbolic Logic*, 1(1):40–41, 1936. Corrected in *J. Symbolic Logic* 1(3):101–102, 1936.

[Ciesielski, 1997] K. Ciesielski. *Set Theory for the Working Mathematician*. Cambridge, 1997.

[Cowell *et al.*, 1999] R. G. Cowell, S. L. Lauritzen, D. J. Spiegelhalter, and A. P. Dawid. *Probabilistic Networks and Expert Systems*. Springer, New York, 1999.

[Culotta and McCallum, 2005] A. Culotta and A. McCallum. Joint deduplication of multiple record types in relational data. In *Proc. 14th Conf. on Information and Knowledge Management*, 2005.

[de Salvo Braz *et al.*, 2005] R. de Salvo Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In *Proc. 19th International Joint Conference on Artificial Intelligence*, pages 1319–1325, 2005.

[Dechter, 1999] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.

[Durrett, 1996] R. Durrett. *Probability: Theory and Examples*. Wadsworth, Belmont, CA, 2nd edition, 1996.

[Elidan and Friedman, 2005] G. Elidan and N. Friedman. Learning hidden variable networks: The information bottleneck approach. *J. Machine Learning Res.*, 6:81–127, 2005.

[Enderton, 2001] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2nd edition, 2001.

[Escobar and West, 1995] M. D. Escobar and M. West. Bayesian density estimation and inference using mixtures. *J. Amer. Stat. Assoc.*, 90(430):577–588, 1995.

[Fellegi and Sunter, 1969] I. Fellegi and A. Sunter. A theory for record linkage. *J. Amer. Stat. Assoc.*, 64:1183–1210, 1969.

[Felsenstein, 2003] J. Felsenstein. *Inferring Phylogenies*. Sinauer, Sunderland, MA, 2003.

[Ferguson, 1983] T. S. Ferguson. Bayesian density estimation by mixtures of normal distributions. In M. H. Rizvi, J. S. Rustagi, and D. Siegmund, editors, *Recent Advances in Statistics: Papers in Honor of Herman Chernoff on His Sixtieth Birthday*, pages 287–302. Academic Press, New York, 1983.

[Fraïssé, 2000] R. Fraïssé. *Theory of Relations*. Elsevier, Amsterdam, revised edition, 2000.

[Franchella, 1997] M. Franchella. On the origins of Dénes König's infinity lemma. *Arch. Hist. Exact Sci.*, 51:3–27, 1997.

[Friedman *et al.*, 1999] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proc. 16th International Joint Conference on Artificial Intelligence*, pages 1300–1307, 1999.

[Fung and Chang, 1990] R. Fung and K.-C. Chang. Weighing and integrating evidence for stochastic simulation in Bayesian networks. In M. Henrion, R. D. Shachter, L. N. Kanal, and J. F. Lemmer, editors, *Uncertainty In Artificial Intelligence 5*, pages 209–219. North-Holland, Amsterdam, 1990.

[Fung and Shachter, 1990] R. M. Fung and R. D. Shachter. Contingent influence diagrams. Working Paper, Dept. of Engineering-Economic Systems, Stanford University, 1990.

[Gaifman, 1964] H. Gaifman. Concerning measures in first order calculi. *Israel J. Math.*, 2:1–18, 1964.

[Gale and Sampson, 1995] W. A. Gale and G. Sampson. Good-Turing frequency estimation without tears. *Quantitative Linguistics*, 2(3):217–237, 1995.

[Geiger and Heckerman, 1996] D. Geiger and D. Heckerman. Knowledge representation and inference in similarity networks and Bayesian multinets. *Artificial Intelligence*, 82(1–2):45–74, 1996.

[Gelman, 1992] A. Gelman. Iterative and non-iterative sampling algorithms. *Comput. Sci. and Stat.*, 24:433–438, 1992.

[Geman and Geman, 1984] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

[Georgii, 1988] H.-O. Georgii. *Gibbs Measures and Phase Transitions*. de Gruyter, Berlin, 1988.

[Getoor *et al.*, 2001] L. Getoor, N. Friedman, D. Koller, and B. Taskar. Learning probabilistic models of relational structure. In *Proc. 18th Int'l Conf. on Machine Learning*, pages 170–177, 2001.

[Giles *et al.*, 1998] C. L. Giles, K. D. Bollacker, and S. Lawrence. CiteSeer: An automatic citation indexing system. In *Proc. 3rd ACM Conf. on Digital Libraries*, pages 89–98, 1998.

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

[Gilks *et al.*, 1996] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, editors. *Markov Chain Monte Carlo in Practice*. Chapman and Hall, London, 1996.

[Haario *et al.*, 2001] H. Haario, E. Saksman, and J. Tamminen. An adaptive Metropolis algorithm. *Bernoulli*, 7:223–242, 2001.

[Halpern, 1990] J. Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46:311–350, 1990.

[Hastings, 1970] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57:97–109, 1970.

[Heckerman *et al.*, 2004] D. Heckerman, C. Meek, and D. Koller. Probabilistic models for relational data. Technical Report MSR-TR-2004-30, Microsoft Research, 2004.

[Henrion, 1988] M. Henrion. Propagating uncertainty in Bayesian networks by probabilistic logic sampling. In J. F. Lemmer and L. N. Kanal, editors, *Uncertainty in Artificial Intelligence 2*, pages 149–163. North-Holland, Amsterdam, 1988.

[Horsch and Poole, 1990] M. C. Horsch and D. Poole. A dynamic approach to probabilistic inference using Bayesian networks. In *Proc. 6th Conf. on Uncertainty in Artificial Intelligence*, pages 155–161, 1990.

[Howard and Matheson, 1984] R. A. Howard and J. E. Matheson. Influence diagrams. In *Readings on the Principles and Applications of Decision Analysis*, volume 2. Strategic Decision Group, Menlo Park, CA, 1984. Reprinted, *Decision Analysis* 2(3):127–143, 2005.

[Ishwaran and James, 2001] H. Ishwaran and L. F. James. Gibbs sampling methods for stick-breaking priors. *J. Amer. Stat. Assoc.*, 96(453):161–173, 2001.

[Jaeger, 1997] M. Jaeger. Relational Bayesian networks. In *Proc. 13th Conf. on Uncertainty in Artificial Intelligence*, pages 266–273, 1997.

[Jaeger, 1998] M. Jaeger. Reasoning about infinite random structures with relational Bayesian networks. In *Proc. 6th Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 570–581, 1998.

[Jain and Neal, 2004] S. Jain and R. M. Neal. A split-merge Markov chain Monte Carlo procedure for the Dirichlet process mixture model. *J. Computational and Graphical Statistics*, 13(1):158–182, 2004.

[Jordan *et al.*, 1999] M. Jordan, Z. Ghahramani, T. Jaakkola, and L. Saul. Introduction to variational methods for graphical models. *Machine Learning*, 37:183–233, 1999.

[Kalmár, 1936] L. Kalmár. Zurückführung des Entscheidungsproblems auf den Fall von Formeln mit einer einzigen, binären Funktionsvariablen. *Compositio mathematica*, 4:137–144, 1936.

[Kersting and De Raedt, 2001a] K. Kersting and L. De Raedt. Adaptive Bayesian logic programs. In *Proc. 11th Int'l Conf. on Inductive Logic Programming*, 2001.

[Kersting and De Raedt, 2001b] K. Kersting and L. De Raedt. Bayesian logic programs. Technical report, Institute for Computer Science, Univ. of Freiburg, Germany, April 2001.

[Kiiveri *et al.*, 1984] H. Kiiveri, T. P. Speed, and J. B. Carlin. Recursive causal models. *J. Austral. Math. Soc. A*, 36:30–52, 1984.

[Koller and Pfeffer, 1998] D. Koller and A. Pfeffer. Probabilistic frame-based systems. In *Proc. 15th AAAI National Conference on Artificial Intelligence*, pages 580–587, 1998.

[König, 1926] D. König. Sur les correspondances multivoques des ensembles. *Fundamenta Mathematicae*, 8:114–134, 1926.

[König, 1927] D. König. Über eine Schlußweise aus dem Endlichen ins Unendliche. *Acta Litterarum ac Scientiarum: Sectio Scientiarum Mathematicarum (Szeged, Hungary)*, 3:121–130, 1927.

[Lafferty *et al.*, 2001] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. 18th Int'l Conf. on Machine Learning*, pages 282–289, 2001.

[Lappin and Leass, 1994] S. Lappin and H. J. Leass. An algorithm for pronominal anaphora resolution. *Computational Linguistics*, 20(4):535–561, 1994.

[Laskey, 2006] K. B. Laskey. First-order Bayesian logic. Technical report, Dept. of Systems Engineering and Operations Research, George Mason Univ., Fairfax, VA, February 2006.

[Lauritzen and Spiegelhalter, 1988] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *J. Royal Stat. Soc. B*, 50:157–224, 1988.

[Lauritzen *et al.*, 1990] S. L. Lauritzen, A. P. Dawid, B. N. Larsen, and H.-G. Leimer. Independence properties of directed Markov fields. *Networks*, 20:491–505, 1990.

[Lawrence *et al.*, 1999a] S. Lawrence, C. L. Giles, and K. Bollacker. Digital libraries and autonomous citation indexing. *IEEE Computer*, 32(6):67–71, 1999.

[Lawrence *et al.*, 1999b] S. Lawrence, C. L. Giles, and K. D. Bollacker. Autonomous citation matching. In *Proc. 3rd Int'l Conf. on Autonomous Agents*, pages 392–393, 1999.

[Li *et al.*, 2004] X. Li, P. Morie, and D. Roth. Robust reading: Identification and tracing of ambiguous names. In *Proc. Human Language Technology Conference and NAACL Annual Meeting*, pages 17–24, 2004.

[Löwenheim, 1915] L. Löwenheim. Über möglichkeiten im relativkalkül. *Mathematische Annalen*, 76, 1915. Trans. in [van Heijenoort, 1967].

[Lukasiewicz and Kern-Isberner, 1999] T. Lukasiewicz and G. Kern-Isberner. Probabilistic logic programming under maximum entropy. In *Proc. 5th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 279–292. Springer, 1999.

[Mathon, 1979] R. Mathon. A note on the graph isomorphism counting problem. *Inform. Process. Lett.*, 8(3):131–132, 1979.

[McCallum and Wellner, 2005] A. McCallum and B. Wellner. Conditional models of identity uncertainty with application to noun coreference. In *Advances in Neural Information Processing Systems 17*. MIT Press, Cambridge, MA, 2005.

[McCallum *et al.*, 2000] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. 6th ACM SIGKDD Int'l Conf. on Knowledge Discovery in Databases*, 2000.

[Metropolis *et al.*, 1953] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *J. Chemical Physics*, 21:1087–1092, 1953.

[Milch and Russell, 2006] B. Milch and S. Russell. General-purpose MCMC inference over relational structures. In *Proc. 22nd Conf. on Uncertainty in Artificial Intelligence*, pages 349–358, 2006.

[Milch *et al.*, 2004] B. Milch, B. Marthi, and S. Russell. BLOG: Relational modeling with unknown objects. In *ICML Workshop on Statistical Relational Learning and Its Connections to Other Fields*, Banff, Alberta, Canada, 2004.

[Milch *et al.*, 2005a] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Proc. 19th International Joint Conference on Artificial Intelligence*, pages 1352–1359, 2005.

[Milch *et al.*, 2005b] B. Milch, B. Marthi, D. Sontag, S. Russell, D. L. Ong, and A. Kolobov. Approximate inference for infinite contingent Bayesian networks. In *Proc. 10th Int'l Workshop on Artificial Intelligence and Statistics*, 2005.

[Mjolsness, 2006] E. Mjolsness. Stochastic process semantics for dynamical grammar syntax: An overview. In *Proc. 9th Int'l Symposium on Artificial Intelligence and Mathematics*, 2006.

[MUC–6, 1995] MUC–6. Coreference resolution task definition (v2.3, 8 Sep 95). In *Proc. 6th Message Understanding Conference*, pages 335–344, 1995.

[Muggleton and Buntine, 1988] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proc. 5th Int'l Conf. on Machine Learning*, pages 339–352, 1988.

[Muggleton, 1996] S. H. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.

[Murphy *et al.*, 1999] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proc. 15th Conf. on Uncertainty in Artificial Intelligence*, pages 467–475, 1999.

[Nash-Williams, 1967] C. St. J. A. Nash-Williams. Infinite graphs — a survey. *J. Combin. Theory*, 3:286–301, 1967.

[Neal, 1991] R. M. Neal. Bayesian mixture modeling by Monte Carlo simulation. Technical Report CRG-TR-91-2, Dept. of Computer Science, Univ. of Toronto, 1991.

[Newcombe *et al.*, 1959] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130:954–959, 1959.

[Ng and Subrahmanian, 1992] R. T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[Ngo and Haddawy, 1997] L. Ngo and P. Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Comp. Sci.*, 171(1–2):147–177, 1997.

[Oh *et al.*, 2004] S. Oh, S. Russell, and S. Sastry. Markov chain Monte Carlo data association for general multiple-target tracking problems. In *Proc. 43rd IEEE Conf. on Decision and Control*, pages 735–742, 2004.

[Otero and Muggleton, 2006] R. Otero and S. Muggleton. On McCarthy's appearance and reality problem. In *Short Papers from the 16th International Conference on Inductive Logic Programming*, 2006.

[Pasula *et al.*, 2003] H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. Identity uncertainty and citation matching. In *Advances in Neural Information Processing Systems 15*. MIT Press, Cambridge, MA, 2003.

[Pasula, 2003] H. Pasula. *Identity Uncertainty*. PhD thesis, Univ. of California at Berkeley, 2003.

[Pearl, 1988] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, San Francisco, revised edition, 1988.

[Pfeffer and Koller, 2000] A. Pfeffer and D. Koller. Semantics and inference for recursive probability models. In *Proc. 17th AAAI National Conference on Artificial Intelligence*, pages 538–544, 2000.

[Pfeffer *et al.*, 1999] A. Pfeffer, D. Koller, B. Milch, and K. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge. In *Proc. 15th Conf. on Uncertainty in Artificial Intelligence*, pages 541–550, 1999.

[Pfeffer, 2000] A. Pfeffer. *Probabilistic Reasoning for Complex Systems*. PhD thesis, Stanford Univ., 2000.

[Pfeffer, 2001] A. Pfeffer. IBAL: A probabilistic rational programming language. In *Proc. 17th International Joint Conference on Artificial Intelligence*, 2001.

[Poole and Zhang, 2003] D. Poole and N. L. Zhang. Exploiting contextual independence in probabilistic inference. *J. Artificial Intelligence Res.*, 18:263–313, 2003.

[Poole, 1993] D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993.

[Poole, 1997] D. Poole. The Independent Choice Logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1–2):5–56, 1997.

[Poole, 2003] D. Poole. First-order probabilistic inference. In *Proc. 18th International Joint Conference on Artificial Intelligence*, pages 985–991, 2003.

[Puech and Muggleton, 2003] A. Puech and S. Muggleton. A comparison of stochastic logic programs and Bayesian logic programs. In *IJCAI Workshop on Learning Statistical Models from Relational Data*, 2003.

[Rabiner, 1989] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[Reiter, 1980] R. Reiter. Equality and domain closure in first-order databases. *Journal of the ACM*, 27:235–249, 1980.

[Revoredo *et al.*, 2006] K. Revoredo, A. Paes, G. Zaverucha, and V. Santos Costa. Combining predicate invention and revision of probabilistic FOL theories. In *Short Papers from the 16th International Conference on Inductive Logic Programming*, 2006.

[Richardson and Domingos, 2006] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

[Richardson and Green, 1997] S. Richardson and P. J. Green. On Bayesian analysis of mixtures with an unknown number of components. *J. Royal Stat. Soc. B*, 59:731–792, 1997.

[Rubinstein, 1981] R. Y. Rubinstein. *Simulation and the Monte Carlo Method*. Wiley, New York, 1981.

[Russell and Norvig, 2003] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Morgan Kaufmann, Upper Saddle River, NJ, 2nd edition, 2003.

[Russell, 2001] S. Russell. Identity uncertainty. In *Proc. 9th Int'l Fuzzy Systems Assoc. World Congress*, 2001.

[Sang *et al.*, 2005] T. Sang, P. Beame, and H. Kautz. Performing Bayesian inference by weighted model counting. In *Proc. 20th AAAI National Conf. on Artificial Intelligence*, pages 475–482, 2005.

[Sato and Kameya, 1997] T. Sato and Y. Kameya. PRISM: A symbolic–statistical modeling language. In *Proc. 15th International Joint Conference on Artificial Intelligence*, pages 1330–1335, 1997.

[Sato and Kameya, 2001] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic–statistical modeling. *J. Artificial Intelligence Res.*, 15:391–454, 2001.

[Shachter and Peot, 1990] R. D. Shachter and M. A. Peot. Simulation approaches to general probabilistic inference on belief networks. In M. Henrion, R. D. Shachter, L. N. Kanal, and J. F. Lemmer, editors, *Uncertainty in Artificial Intelligence 5*, pages 221–231. North-Holland, Amsterdam, 1990.

[Shachter *et al.*, 1990] R. D. Shachter, B. D'Ambrosio, and B. A. Del Favero. Symbolic probabilistic inference in belief networks. In *Proc. 8th AAAI National Conference on Artificial Intelligence*, pages 126–131, 1990.

[Shachter, 1986] R. D. Shachter. Evaluating influence diagrams. *Operations Research*, 34(6):871–882, 1986.

[Singla and Domingos, 2005] P. Singla and P. Domingos. Object identification with attribute-mediated dependences. In *Proc. 9th European Conf. on Principles and Practice of Knowledge Discovery in Databases*, pages 297–308, 2005.

[Singla and Domingos, 2006] P. Singla and P. Domingos. Entity resolution with Markov logic. In *Proc. IEEE Int'l Conf. on Data Mining*, 2006.

[Sittler, 1964] R. W. Sittler. An optimal data association problem in surveillance theory. *IEEE Trans. Military Electronics*, MIL-8:125–139, 1964.

[Skolem, 1920] Th. Skolem. Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit und Beweisbarkeit mathematischen Sätze nebst einem Theoreme über dichte Mengen. *Videnskapsselskapets skrifter, 1: Mathematik-naturvidenskabelig klasse (Kristiana, Norway)*, 4, 1920. Relevant portion trans. in [van Heijenoort, 1967].

[Soon *et al.*, 2001] W. M. Soon, H. T. Ng, and D. C. Y. Lim. A machine learning approach to coreference resolution of noun phrases. *Computational Linguistics*, 27(4):521–544, 2001.

[Taskar *et al.*, 2002] B. Taskar, P. Abbeel, and D. Koller. Discriminative probabilistic models for relational data. In *Proc. 18th Conf. on Uncertainty in Artificial Intelligence*, pages 485–492, 2002.

[Thomas *et al.*, 1992] A. Thomas, D. Spiegelhalter, and W. Gilks. BUGS: A program to perform Bayesian inference using Gibbs sampling. In J. Bernardo, J. Berger, A. Dawid, and A. Smith, editors, *Bayesian Statistics 4*. Oxford Univ. Press, 1992.

[Tierney, 1996] L. Tierney. Introduction to general state-space Markov chain theory. In Gilks et al. [1996], pages 59–74.

[Tu and Zhu, 2002] Z. W. Tu and S. C. Zhu. Image segmentation by data-driven Markov chain Monte Carlo. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 24(5):657–673, 2002.

[van Heijenoort, 1967] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic 1879–1931*. Harvard Univ. Press, Cambridge, MA, 1967.

[Vennekens *et al.*, 2004] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *Proc. Int'l Conf. on Logic Programming*, pages 431–445, 2004.

[Wei and Tanner, 1990] G. C. G. Wei and M. A. Tanner. A Monte Carlo implementation of the EM algorithm and the poor man's data augmentation algorithms. *J. Amer. Stat. Assoc.*, 85:699–704, 1990.

[Wellner *et al.*, 2004] B. Wellner, A. McCallum, F. Peng, and M. Hay. An integrated, conditional model of information extraction and coreference with application to citation matching. In *Proc. 20th Conf. on Uncertainty in Artificial Intelligence*, 2004.

[Winkler, 2006] W. E. Winkler. Overview of record linkage and current research directions. Research Report 2006–02, Statistical Research Div., U.S. Census Bureau, Washington, DC, 2006.

[Xing *et al.*, 2003] E. P. Xing, M. I. Jordan, and S. Russell. A generalized mean field algorithm for variational inference in exponential families. In *Proc. 19th Conf. on Uncertainty in Artificial Intelligence*, pages 583–591, 2003.

[Zhang and Poole, 1994] N. L. Zhang and D. Poole. A simple approach to Bayesian network computations. In *Proc. 10th Canadian Conference on Artificial Intelligence*, pages 171–178, 1994.