Fabrizio Riguzzi Joost Vennekens (Eds.)

PLP 2015
Probabilistic Logic Programming

2nd International Workshop on Probabilistic Logic Programming co-located with 31st International Conference on Logic Programming (ICLP 2015)

Cork, Ireland, August 31, 2015

Proceedings

CEUR-WS.org/Vol-1413

urn:nbn:de:0074-1413-7

http://ceur-ws.org/Vol-1413

Editors' addresses:
Fabrizio Riguzzi
Department of Mathematics and Computer Science, University of Ferrara
Via Saragat 1, 44122, Ferrara, Italy

fabrizio.riguzzi@unife.it

Joost Vennekens
Department of Computer Science, KU Leuven
Jan De Nayerlaan 5, 2860 Sint-Katelijne-Waver, Belgium

joost.vennekens@cs.kuleuven.be

Preface

This is the proceedings of the Second Workshop on Probabilistic Logic Programming (PLP 2015), which was held on August 31st 2015 in Cork, Ireland, as a workshop of the 31st International Conference on Logic Programming (ICLP 2015).

Eight papers were submitted to the workshop. Each submission was reviewed by three members of the program committee. All submitted papers were of sufficiently high quality to be accepted to the workshop. In addition, the workshop also had invited talks by Angelika Kimmig (KU Leuven) and Nicos Angelopoulos (Imperial College, London).

This workshop is the second edition in what we hope will be a long series. The first edition was held in 2014 in Vienna, Austria, also as part of the ICLP conference. More information about the current edition, the previous edition, and future editions can be found at the following website:

`http://stoics.org.uk/plp/`

We would like to thank all authors who submitted papers, all program committee members and all reviewers for their efforts. In addition, we are also grateful to the organisers of ICLP and, in particular, to Mats Carlsson, ICLP's Workshop Chair.

July 2015 Fabrizio Riguzzi, Joost Vennekens

Organizing Committee

Fabrizio Riguzzi	University of Ferrara, Italy
Joost Vennekens	KU Leuven, Belgium

Program Committee

Nicos Angelopoulos	Imperial College, UK
Elena Bellodi	University of Ferrara, Italy
James Cussens	University of York, UK
Nicola Di Mauro	Università di Bari, Italy
Arjen Hommersom	University of Nijmegen, The Netherlands
Angelika Kimmig	KU Leuven, Belgium
Wannes Meert	KU Leuven, Belgium
Aline Paes	Universidade Federal Fluminense, Brasil
David Poole	University of British Columbia, Canada
C. R. Ramakrishnan	University at Stony Brook, US
Fabrizio Riguzzi	University of Ferrara, Italy
Terrance Swift	Coherent Knowledge Systems, US
Christian Theil Have	Copenhagen University, Denmark
Joost Vennekens	KU Leuven, Belgium
Herbert Wiklicky	Imperial College, UK

Contents

Presented Papers which Appear Elsewhere

BIMS: for Bayesian inference of model structure

Nicos Angelopoulos

Department of Surgery and Cancer, Division of Cancer, Imperial College London, Hammersmith Hospital Campus, London, UK.

MCMC methods in probabilistic logic programming settings are gaining popularity and a number of different approaches have been proposed recently. We discuss theoretical results and experiences with applications of one of the first approaches in the field. The knowledge representation capabilities of the underlying language, which are less well documented in the literature, are discussed, as well as the machine learning applications of the general framework, which have been presented in a number of publications. We focus on how to express Bayesian prior knowledge in this formalism, and show how it can be used to define generative priors over statistical model spaces: Bayesian networks and classification and regression trees. Finally, we discuss a Metropolis-Hastings algorithm that can take advantage of the defined priors and its application to real-world machine learning tasks. Details of the associated publicly available software are also discussed.

Probabilistic (logic) programming: opportunities and challenges

Angelika Kimmig

Department of Computer Science, KU Leuven, Belgium

Probabilistic logic programming has a long tradition dating back at least to the seminal works of David Poole and Taisuke Sato in the early 90s. Still, the enormous recent interest in probabilistic programming is very much focused on other programming paradigms. This talk will explore commonalities as well as differences between PLP and other PP languages, focusing on the opportunities and challenges they provide.

Advances in integrating statistical inference

Nicos Angelopoulos[1] Samer Abdallah[2] and Georgios Giamas[1]

[1] Department of Surgery and Cancer, Division of Cancer, Imperial College London, Hammersmith Hospital Campus, Du Cane Road, London W12 ONN, UK.
[2] Department of Computer Science, University College London Gower Street, London WC1E 6BT, UK.

Abstract. We present recent developments on the syntax of *Real*, a library for interfacing two Prolog systems to the statistical language *R*. We focus on the changes in Prolog syntax within SWI-Prolog that accommodate greater syntactic integration, enhanced user experience and improved features for web-services. We recount the full syntax and functionality of *Real* as well as presenting sister packages which include Prolog code interfacing a number of common and useful tasks that can be delegated to *R*. We argue that *Real* is a powerful extension to logic programming, providing access to a popular statistical system that has complementary strengths in areas such as machine learning, statistical inference and visualisation. Furthermore, *Real* has a central role to play in the uptake of computational biology and bioinformatics as application areas for research in logic programming.

1 Introduction

Real is a low level interface between Prolog and *R* (Angelopoulos et al., 2013). It enables the user to call *R* functions on Prolog data and communicate the results back to the logic system. The library works on two open source systems: YAP (Costa et al., 2012) and SWI-Prolog (Wielemaker et al., 2012) whose *C* language interface is compatible (Wielemaker and Costa, 2011). Since its first introduction *Real* has evolved and has exerted some influence in advances to Prolog syntax. Furthermore, it has been used in a number of projects and in the process acquired a number of sister libraries that depend on it to deliver Prolog interfaces to useful tasks that can be best be dealt by existing *R* code. *Real* has thus be shown to be a useful and well integrated Prolog library that can provide access to the wealth of open source code available in *R* which is often accompanied by published scientific papers.

Here we focus on describing the full syntax of *Real 1.4* and its role in recent developments with syntactic changes in *SWI-7*. The changes in both systems have made the integration of *R* code into Prolog more natural and unobtrusive. Changes in the library itself had to be made to accommodate transition to the new Prolog syntax while preserving compatibility with traditional implementations.

Real gives access to *R* libraries that can complement Prolog's weaknesses in areas such as statistical inference and visualisation. With the library installed it

is straight forward with a basic grasp of *R* to call its functions on Prolog data. However for users with no prior exposure to *R* there still might be a barrier. To address this, and in order to increase general usability of the library a number of sister packages have been developed. We highlight some of the predicates that enable access to *R* code without any knowledge of *R*.

Central application areas in the inception of *Real* and its recent advances, has been the areas of bioinformatics and computational biology. The sister libraries we describe here have evolved in addressing real world bioinformatics tasks in the context of a variety of projects: (Zhang et al., 2015; MacIntyre et al., 2015; Stebbing et al., 2015). The main thesis of this paper is that Prolog can play a central role as a unifying platform in research in bioinformatics, taking advantage of its strong grip in knowledge representation and reasoning and in combinations with recent advances with *Real* and web programming (Wielemaker et al., 2008; Lager and Wielemaker, 2014).

2 *Real*

Here we first present the innovations of *Real 1.4* before we summarise its overall syntax and usage with particular focus on new features.

2.1 Innovations

In terms of syntax, *Real* faced three major clashes between Prolog notation and syntax acceptable to *R*. Those were the use of '.' in *R* identifiers, the use of double quotes (' " ') to represent strings and the representation of terms with 0 arity '$foo()$'. In previous versions the library was able to bypass those by employing a number of indirect techniques concentrating on keeping as faithful as possible to the original syntax. Briefly,

- operator '..' was used to construct arity 2 terms that were behind the scenes converted to a Prolog atom interpreted as an *R* identifier (Prolog term $my..variable$ was translated to *R* variable $my.variable$, $my..variable \rightarrow my.variable$).
- operator + on non numerical values was used to convert atoms and code lists to strings ($+foo \rightarrow "foo"$)
- with the newly, at the time, introduced block operator '()' it was possible to parse $foo`()'$ as $foo()$

With *Real* in mind, *SWI-7* (Wielemaker, 2014) introduced syntax that legalised all of the above constructs, as well as the implementation of lists as primary data structures (as oppose to ./2 terms). Dots in atoms and the use of double quotes are now controlled by global flags, the former's default being off and the latter's being on. *Real* has been adapted to utilise the new changes in a backwards compatible manner. All of the following are now valid *Real* syntax mapping to the corresponding *R* constructs, proviso of the appropriate global flags been enabled,

- func.foo(a,b,c)
- write.csv("to_file.csv", x)
- foo()

Under the bonnet, list representations were additionally generalised to accommodate the new data type. Also in the *C* interface, *Real 1.4* includes improvements in that it can be employed within a web-service, thus allowing the *R*-server thread to be an arbitrary one. This is of particular interest, as in itself *R* is single threaded.

A final innovation at the syntactic level has been the introduction of 'NA' values in the interface. In *R*, NA values stand for not available or unknown value placeholders. Prolog does not internally support such values, but the interface enables mapping of such values within arithmetic vectors and matrices to '$NaN'. When passing numeric data from Prolog to *R* in addition to $NaN, the empty atom ('') is also translated to *R* 's NA value.

Taken together these innovations allow a tighter and smoother integration of *R* code and enable Prolog programmers to tap in the wealth of statistical functions implemented in *R*.

2.2 Communication with *R*

The bulk of the communication with *R* is via a single predicate $\leftarrow /2$ which is also defined as an infix operator. This is an alternative assignment operator in *R*. Within *Real* it can be used to transfer data between *R* and Prolog, to apply, in an in-line fashion, *R* functions to Prolog data as well as destructively assigning values to *R* variables. Disambiguation clearly distinguishes the different modes, which can be summarised by:

$$+Rexpr \leftarrow +Rexpr$$
$$-PlVar \leftarrow +Rexpr$$
$$+Rexpr \leftarrow +PlData$$

When the LHS of the operator is a uninstantiated variable, the second mode is assumed, where the value of *Rexpr* is passed to $PlVar$ after it has been evaluated in *R*. When the RHS is a c/n term or a list then the third mode is applied and the data in the RHS is transferred to the LHS *Rexpr* (usually an *R* variable).

The following examples show how to: transfer Prolog data to *R* and back (1), transfer Prolog data to *R* and get the result of applying a function to the data in the new *R* variable (2) and demonstrating how to apply an *R* function on Prolog data without the use of an explicit *R* variable (3).

$$?- a \leftarrow [1,2,3], A \leftarrow a. \tag{1}$$

$$A = [1,2,3].$$

Indicator	Operator Symbol		Description
r/1	`<-`	$\leftarrow$	evaluate R expression (no return value)
r/2	`<-`	$\leftarrow$	main communication to R library
r_new/1	`<<-`	$\twoheadleftarrow$	argument is a fresh variable
`<<-`/2	`<<-`	$\twoheadleftarrow$	r/2 but with error if R variable exists
r_call/2	`<-C++O`	$\leftarrow$ ++	r/1,2 with options (O)
r_library/1			load R library in a hookable manner
r_start/0			start the connection to R
r_stop/0			stop the connection to R
r_remove/1			remove R variable
r_thread_loop/0			start an R thread server
r_serve/0			serve all R expressions on queue thread

Table 1. Library's main predicates

$$?- a \leftarrow [1,2,3], Mean \leftarrow mean(a). \tag{2}$$

$$Mean = 2.0.$$

$$?- Mean \leftarrow mean([1,2,3]). \tag{3}$$

$$Mean = 2.0.$$

2.3 Real's predicates

Real 1.4 adopts the convention of a uniform prefix to all the library predicates. The full list of *Real*'s predicates along with the associated operators and brief descriptions are shown in Table 1. New additions include a hookable locator for R libraries, web server support, intuitive syntax for non-destructive assignment and a generic predicate for mixing Prolog and R options and directing output to graphic devices.

With new predicate *r_library/1* the user can load the standard R libraries in their local installation. In addition, the predicate can be directed to user specified locations where local, possibly, changed sources of such libraries can be loaded preferentially. The flexibility allows for (a) specific code to be loaded only known to *Real* thus living the remainder of the R installation intact, and (b) user code that can be made available and can work either with the distributed version while having extra functionality when used with the altered sources.

Real is inherently single threaded. To support the use of *Real* in multi-threaded applications, in particular in web servers built on SWI Prolog's HTTP libraries (Wielemaker et al., 2008), *Real 1.4* allows a single designated *Real* server thread to be started, which then takes over the task of executing or evaluating

R commands or expressions. Then, when the *←/1* and *←/2* predicates are used on any other thread, the requests are redirected to the *Real* server thread and the results awaited. Communication is handled synchronously using SWI Prolog queues.

This system was implemented to support an application in the area of large scale computational musicology, the *Digital Music Laboratory*, which is built on SWI-Prolog's semantic web server *Cliopatria*. Here, *Real* is used both for general numerical computations and the generation of high-quality scalable vector graphics. In comparison with previous versions of the system which used Matlab's engine API to communicate with a separate *Matlab* process, the lower overhead of communicating with *Real*'s in-process embedded *R* yields much better performance when numerous relatively small computations are required.

As *R* supports destructive assignment, it can be the case that the programmer might unwittingly overwrite variables already in the working space. To ease and provide visual cues of the fact that a variable is fresh in a specific context, we introduced operators *↞/2* and *↞/1* and predicate *r_new/1*. The first ensures that its first argument (an *R* variable) did not exist prior to assigning to it some new values. The second removes its arguments from the *R* work-space and the third fails if its argument is already a known *R* variable.

Integral to the *R* language design and practice is the use of options that control the details of function calls. These are = pairs of argument name to values, which more often than not do not have to be present at invocation. When not present, default values supplied by the function developers are used. Similarly but not as widely used is the use of list of terms that control calls to Prolog predicates. By convention an options list is placed at the last argument of a predicate and commonly contains a number of single arity terms. *Real* now provides a uniform way to marry the two conventions and a flexible way of handling options addressed to Prolog predicates accessing *R* functions. In addition, a number of standard tasks have been incorporated to a new interface predicate:

$$r\_call(Func, Opts).$$

which can also be accessed as

$$\leftarrow Func \;\; +\!\!+ \;\; Opts$$

$Func$ is a compound term which is translated to an *R* function call and $Opts$ can be a combination of: (a) *=/2* terms, which are added to $Func$, (b) options controlling *r_call/2*'s own execution and (c) Prolog style options which can influence the caller's behaviour but are ignored in the *R* call. Some of *r_call/2* options are:

rvar(Rvar) when given call becomes: $Rvar \leftarrow Fcall$
rmv(Rmv=false) removes *Rvar* after end of call
stem(Stem=real_plot) stem to use for output files
outputs(Outs=false) a list of output devices
debug(Dbg=false) sets debug(real) for the duration of call
fcall(FinCall) returns the term constructed after *=/2* additions
post_call(Post) call this after the function call

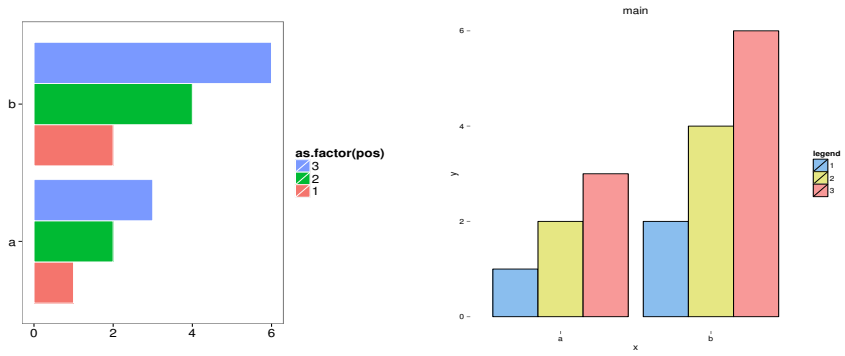


Fig. 1. ggplot2 based bar plots. Left: with default options. Right: a number of options have altered elements of the plot.

3 Associated packages

3.1 *b_real*

b_real is a library based on *Real* which contains a collection of predicates that aim to provide a Prolog based interface to a number of simple tasks. The target audience is Prolog users that have no previous experience with *R*. The predicates described here can use the basic functionality of the underlying *R* functions and can adjust some of the behaviour entirely in Prolog, while allowing arbitrary option passing to users with some familiarity with *R*.

Bar plots are basic plots that can present comparative information in a intuitive manner. Here we present a Prolog interface to *ggplot2* (Wickham, 2009). In its most general form, predicate *gg_bar_plot/2* displays a number of grouped measurements such as, for instance, the cpu-timings of a number of machine learning algorithms ran on a number of datasets. The following query, produces the plot in the LHS of Fig. 3.1.

$$?- Pairs = [a-[1,2,3], b-[2,4,6]],\ gg\_bar\_plot(Pairs, [\,]). \tag{4}$$

ggplot2 is a complex piece of software able to display many types of plots while *gg_bar_plot/2* only accessing the bar plotting part. Within this, a number of plot elements can be controlled with Prolog options passed in the second argument. The following query changes elements such as the colour of the drawing pen (black) the labels (x,y and main), legend title and fill colours, producing the plot in the RHS of Fig. 3.1.

$$\begin{aligned} ?- Pairs &= [a-[1,2,3], b-[2,4,6]], \\ Opts &= [\, geom\_bar\_draw\_colour(black), \\ & fill\_colours([\text{"}skyblue2\text{"}, \text{"}khaki2\text{"}, \text{"}\#FB9A99\text{"}]), \end{aligned} \tag{5}$$

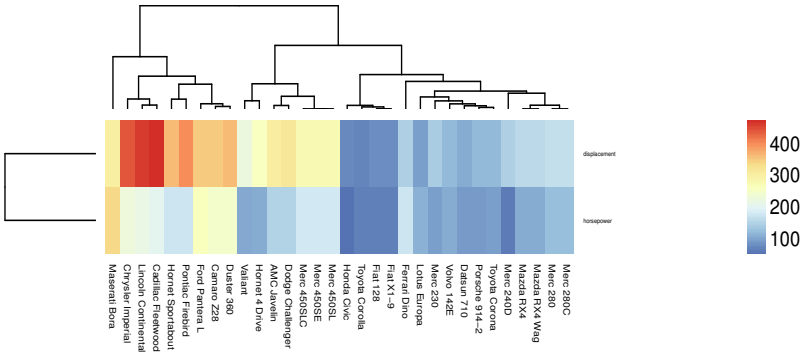


Fig. 2. Heatmap generation with aheatmap() from package NMF.

$$flip(false), labels(x, y, main),$$
$$legend\_title(legend)],$$
$$gg\_bar\_plot(Pairs, Opts).$$

Heatmap functions are ubiquitous in *R*. *b_real* provides a Prolog interface to the *aheatmap* library. In addition to some simple option mapping *aheatmap/2* provides polymorphic support for the first argument which could be a matrix *R* variable or a Prolog representation of one. The following code uses the *mtcars* example dataset, from which it plots a heatmap of two variables: *hp* (horsepower) and *disp* (displacement).

$$\begin{aligned} ?- MtC \leftarrow as.list(mtcars), memberchk(hp = HP, MtC), \\ memberchk(disp = Disp, MtC), x \leftarrow [HP, Disp], \\ rownames(x) \leftarrow c(\text{"}horsepower\text{"}, \text{"}displacement\text{"}), \\ <- aheatmap(x). \end{aligned} \tag{6}$$

3.2 wgraph

R has a number of plotting functions for drawing graphs formed of nodes and edges. Two of these are *igraph()* and *qgraph()*. The latter being based on the former with some extra options and facilities for grouping nodes. The Prolog pack *wgraph* provides a uniform Prolog interface to these *R* libraries. A plot

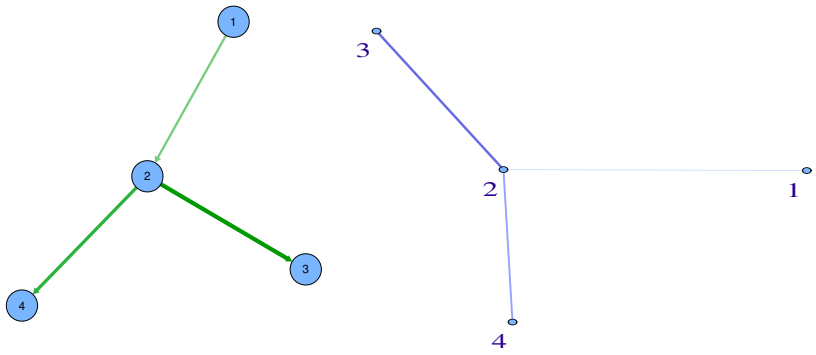


Fig. 3. Graphs generated by *wgraph_plot/2*. Left: plot uses default rendering with *qgraph()* call. Right: render changed to *igraph()* and a number of options specialised the output.

with the default renderings can be easily drawn from a list representing the graph connections and the weights on the edges:

$$\begin{aligned} ?- G = [1-2:200, 2-3:400, 2-4:300], \\ wgraph\_plot(G, [\,]). \end{aligned} \tag{7}$$

A set of Prolog options that control the choice of the drawing function and basic parameters of the graph, and which work irrespective of the drawing function can be provided in the second argument of *wgraph_plot/2*. In the following example *igraph()* is passed the size of nodes to use, the degree at which the node labels should be displayed and the distance of the label from the node edge. The resulting graph is shown in the RHS of Fig. 3.

$$\begin{aligned} ?- G = [1-2:200, 2-3:400, 2-4:300], \\ Opts = [\,plotter(igraph), label\_distance(-1), \\ label\_degree(2), node\_size(4)\,], \\ wgraph\_plot(G, Opts). \end{aligned} \tag{8}$$

3.3 Availability

The three libraries discussed here, (*Real*, *b_real* and *wgraph*) are available as SWI-Prolog packages[3] which can be installed easily from within SWI-Prolog. To download and install *Real* the user needs to query with:

$$?- install\_pack(real). \tag{9}$$

[3] `http://swi-prolog.org/pack/list`

4 Conclusions

We presented a number of recent advances in *Real* and in particular shown how developments in Prolog syntax have made *Real* syntax blend naturally into Prolog code. The resulting syntax provides a powerful platform for accessing the extensive collection of freely available *R* code. As a consequence *Real* can have a strong positive influence into the penetration of Prolog to new application areas such as bioinformatics and machine learning. With version 1.4 *Real* has reached a new level of maturity including facilities for using *R* in web-servers. In addition we highlighted some predicates from two sister packages. As with *Real* itself, these are freely available and can be easily installed via the SWI-Prolog package manager. In the future we plan to work towards suggesting internal ways for Prolog to work better, or more confluent to *R*, with *NA* values and infinity.

Real has been used in a number of projects in the area of bioinformatics and has a steady stream of downloads via SWI-Prolog's package manager. With the enhanced level of integration, *Real* is becoming a powerful hybrid programming language.

Bibliography

Nicos Angelopoulos, Vitor Santos Costa, Joao Azevedo, Jan Wielemaker, Rui Camacho, and Lodewyk Wessels. Integrative functional statistics in logic programming. In *Proc. of Practical Aspects of Declarative Languages*, volume 7752 of *LNCS*, pages 190–205, Rome, Italy, Jan. 2013. URL `http://stoics.org.uk/~nicos/sware/real/`.

Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12:5–34, 1 2012. ISSN 1475-3081.

Torbjorn Lager and Jan Wielemaker. Pengines: Web logic programming made easy. In *International Conference of Logic Programming*, 2014.

David MacIntyre, Manju Chandiramani, Yun S Lee, Lindsay Kindinger, Ann Smith, Nicos Angelopoulos, Benjamin C. Lehne, Shankari Arulkumaran, Richard Brown, Tiong Ghee Teoh, Elaine Holmes, Jeremy K. Nicholson, Julian Marchesi, and Phillip R. Bennett. The vaginal microbiome during pregnancy and the postpartum period in a european population. *Scientific Reports*, 5:Article number: 8988, 2015. URL `http://www.nature.com/srep/2015/150311/srep08988/full/srep08988.html`.

Justin Stebbing, Hua Zhang, , Yichen Xu, Adam Sanit Nicos Angelopoulos, and Georgios Giamas. Global mapping of tyrosine kinase signalling. *Journal Title*, 2015. Accepted for publication.

Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. ISBN 978-0-387-98140-6. URL `http://had.co.nz/ggplot2/book`.

Jan Wielemaker. SWI-Prolog ODBC interface, 2014. URL `http://www.swi-prolog.org/pldoc/package/odbc.html`.

Jan Wielemaker and Vítor Santos Costa. On the portability of Prolog applications. In *Practical aspects of Declarative Languages*, pages 69–83, 2011.

Jan Wielemaker, Zhisheng Huang, and Lourens van der Meij. SWI-Prolog and the web. *TPLP*, 8(3):363–392, 2008.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. ISSN 1471-0684.

Hua Zhang, Nicos Angelopoulos, Yichen Xu, Arnhild Grothey, Joao Nunes, Justin Stebbing, and Georgios Giamas. Proteomic profile of KSR1-regulated signaling in response to genotoxic agents in breast cancer. *Breast Cancer Research and Treatment*, 2015. URL `http://link.springer.com/article/10.1007/s10549-015-3443-y`.

Towards a General Framework for Actual Causation Using CP-logic

Sander Beckers and Joost Vennekens

Dept. Computer Science, Campus De Nayer
KU Leuven - University of Leuven
{Sander.Beckers, Joost.Vennekens}@cs.kuleuven.be

Abstract. Since Pearl's seminal work on providing a formal language for causality, the subject has garnered a lot of interest among philosophers and researchers in artificial intelligence alike. One of the most debated topics in this context is the notion of actual causation, which concerns itself with specific – as opposed to general – causal claims. The search for a proper formal definition of actual causation has evolved into a controversial debate, that is pervaded with ambiguities and confusion. The goal of our research is twofold. First, we wish to provide a clear way to compare competing definitions. Second, we want to improve upon these definitions so they can be applied to a more diverse range of instances, including non-deterministic ones. To achieve these goals we provide a general, abstract definition of actual causation, formulated in the context of the expressive language of CP-logic (Causal Probabilistic logic). We will then show that three recent definitions by Ned Hall (originally formulated for structural models) and a definition of our own (formulated for CP-logic directly) can be viewed and directly compared as instantiations of this abstract definition, which also allows them to deal with a broader range of examples.

Keywords: actual causation, CP-logic, counterfactual dependence

1 Introduction

Suppose we know the causal laws that govern some domain, and that we then observe a story that takes place in this domain; when should we now say that, in this particular story, one event caused another? Ever since [10] first analyzed this problem of actual causation (a.k.a. token causation) in terms of counterfactual dependence, philosophers and researchers from the AI community alike have been trying to improve on his attempt. Following [11], structural equations have become a popular formal framework for this [8, 9, 5, 7]. A notable exception is the work of Ned Hall, who has extensively critiziced the privileged role of structural equations for causal modelling, as well as the definitions that have been expressed with it. He has proposed several definitions himself [2–4], the latest of which is a sophisticated attempt to overcome the flaws he observes in those that rely too

heavily on structural equations. We have developed a definition of our own in [1, 13], within the framework of CP-logic (Causal Probabilistic logic).

The relation between these different approaches is currently not well understood. Indeed, they are all expressed using different formalisms (e.g., neuron diagrams, structural equations, CP-logic, or just natural language). Therefore, comparisons between them are limited to verifying on which examples they (dis)agree. In this paper, we work towards a remedy for this situation. We will present a general, parametrized definition of actual causation in the context of the expressive language of CP-logic. Exploiting the fact that neuron diagrams and structural equations can both be reduced to CP-logic, we will then show that our definition and three definitions by Ned Hall can be seen as particular instantiations of this parametrized definition. This immediately provides a clear, conceptual picture of the similarities and differences between these approaches. Our analysis thus allows for a formal and fundamental comparison between them.

This general framework for comparing different approaches to actual causation is the main contribution of this paper. In addition, placing existing approaches in this framework may make it easier to improve/extend them. Our versions of Hall's definitions illustrate this, as their scope is expanded to also include non-deterministic examples, and cases of causation by omission. Further, our formulations prove to be simpler than the original ones and their application becomes more straightforward. While our ambition is to work towards a framework that encompasses a large variety of approaches to actual causation, this goal is obviously infeasible within the scope of a single paper. We have therefore chosen to focus most of our attention on Hall, because his work is both among the most refined and most influential in this field; in addition, it is also representative for a larger body of work in the counterfactual tradition.

We first introduce the CP-logic language in Section 2. In Section 3, a general definition of actual causation is first presented, and then instantiated into four concrete definitions. Section 4 offers a succinct representation of all these definitions, and an illustration of how they compare to each other.

2 CP-logic

We give a short, informal introduction to CP-logic. A detailed description can be found in [14]. The basic syntactical unit of CP-logic is a CP-law, which takes the general form $Head \leftarrow Body$. The body can in general consist of any first-order logic formula. However, in this paper, we restrict our attention to conjunctions of ground literals. The head contains a disjunction of atoms annotated with probabilities, representing the possible effects of this law. When the probabilities in a head do not add up to one, we implicitly assume an *empty* disjunct, annotated with the remaining probability.

Each CP-law models a specific *causal mechanism*. Informally, if the $Body$ of the law is satisfied, then at some point it will be applied, meaning one of the disjuncts in the $Head$ is chosen, each with their respective probabilities. If a disjunct is chosen containing an atom that is not yet **true**, then this law causes

it to become **true**; otherwise, the law has no effect. A finite set of such CP-laws forms a CP-theory, and represents the causal structure of the domain at hand. The domain unfolds by laws being applied one after another, where multiple orders are often possible, and each law is applied at most once. We illustrate with an example from [3]:

> Suzy and Billy each decide to throw a rock at a bottle. When Suzy does so, her aim is accurate with probability 0.9. Billy's aim is slightly worse, namely 0.8. If a rock hits it, the bottle breaks.

This small causal domain can be expressed by the following CP-theory T:

$$Throws(Suzy) \leftarrow . \quad (1) \qquad (Breaks : 0.9) \leftarrow Throws(Suzy). \quad (3)$$
$$Throws(Billy) \leftarrow . \quad (2) \qquad (Breaks : 0.8) \leftarrow Throws(Billy). \quad (4)$$

The first two laws are *vacuous* (i.e., they will be applied in every story) and *deterministic* (i.e., they have only one possible outcome, where we leave implicit the probability 1). The last two laws are *non-deterministic*, causing either the bottle to break or nothing at all.

The given theory summarizes all possible *stories* that can take place in this model. For example, it allows for the story consisting of the following chain of events:

> Suzy and Billy both throw a rock at a bottle. Suzy's rock gets there first, shattering the bottle. However Billy's throw was also accurate, and would have shattered the bottle had it not been preempted by Suzy's throw.

To formalize this idea, the semantics of CP-logic uses *probability trees* [12]. For this example, one such tree is shown in Figure 1. Here, each node represents a state of the domain, which is characterized by an assignment of truth values to the atomic formulas, in this case $Throws(Suzy)$, $Throws(Billy)$ and $Breaks$. In the initial state of the domain (the root node), all atoms are assigned their *default* value **false**. In this example, the bottle is initially unbroken and the rocks are still in Billy and Suzy's hands. The children of a node x are the result of the application of a law: each edge (x, y) corresponds to a specific disjunct that was chosen from the head of the law that was applied in node x. In this particular case, law (1) is applied first, so the assignment in the child-node is obtained by setting $Throws(Suzy)$ to **true**, its *deviant* value. The third state has two child-nodes, corresponding to law (3) being applied and either breaking the bottle (left child) or not (right child). The leftmost branch is thus the formal counterpart of the above story, where the last edge represents the fact that Billy's throw was also accurate, even though there was no bottle left to break. A branch ends when no more laws can be applied.

A probability tree of a theory T in CP-logic defines an *a priori* probability distribution P_T over all things that might happen in this domain, which can be read off the leaf nodes of the branches by multiplying the probabilities on

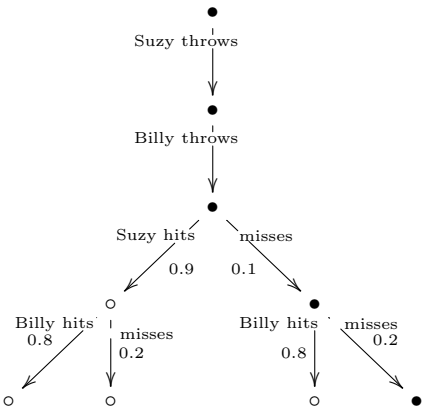


Fig. 1: Probability tree for Suzy and Billy.

the edges. For instance, the probability of the bottle breaking is the sum of the probabilities of the leaves in which *Breaks* is **true** – the white circles in Figure 1 – giving 0.98. We have shown here only one such probability tree, but we can construct others as well by applying the laws in different orders.

An important property however is that all trees defined by the same theory result in the same probability distribution. Thus even though the order in a branch can capture temporal properties of the corresponding story – which play a role in deciding actual causation – it does not affect the resulting assignment. To ensure that this property holds even when there are bodies containing negative literals, CP-logic makes use of a probabilistic variant of the well-founded semantics [14]. Simply put, this means the condition for a law to be applied in a node is not merely that its body is currently satisfied, but that it will remain so. This implies that a negated atom in a body should not only be currently assigned **false**, but actually has to have become impossible, so that it will remain **false** through to the end-state. For atoms currently assigned **true**, it always holds that they remain **true**, hence here there is no problem.

Counterfactual Probabilities In the context of structural equations, [11] studies counterfactuals and shows how they can be evaluated by means of a syntactic transformation. In their study of actual causation and explanations, [6, p. 27] also define counterfactual probabilities (i.e., the probability that some event would have had in a counterfactual situation). [15] present an equivalent method for evaluating counterfactual probabilities in CP-logic, also making use of syntactic transformations.

Assume we have a branch b of a probability tree of some theory T. To make T deterministic *in accordance with the choices made in* b, we transform T into T^b by replacing the heads of the laws that were applied in b with the disjuncts

that were chosen from those heads in b. For example, if we take as branch b the previous story, then T^b would be:

$$Throws(Suzy) \leftarrow . \qquad Breaks \leftarrow Throws(Suzy).$$
$$Throws(Billy) \leftarrow . \qquad Breaks \leftarrow Throws(Billy).$$

We will use Pearl's $do()$-operator to indicate an intervention [11]. The intervention on a theory T that ensures variable C remains false, denoted by $do(\neg C)$, removes C from the head of any law in which it occurs, yielding $T|do(\neg C)$. For example, to prevent Suzy from throwing, the resulting theory $T|do(\neg Throws(Suzy))$ is given by:

$$\leftarrow . \qquad (Breaks : 0.9) \leftarrow Throws(Suzy).$$
$$Throws(Billy) \leftarrow . \qquad (Breaks : 0.8) \leftarrow Throws(Billy).$$

Laws with an empty head are ineffective, and can thus simply be omitted. The analogous operation $do(C)$ on a theory T corresponds to adding the deterministic law $C \leftarrow$.

With this in hand, we can now evaluate a Pearl-style counterfactual probability "given that b in fact occurred, the probability that $\neg E$ would have occurred if $\neg C$ had been the case" as $P_{T^b}(\neg E|do(\neg C))$.

3 Defining Actual Causation Using CP-logic

We now formulate a general, parametrized definition of actual causation, which can accommodate several concrete definitions by filling in details that we first leave open. We demonstrate this using definitions by Hall and one by ourselves. For the rest of the paper, we assume that we are given a CP-theory T, an actual story b in which both C and E occurred, and we are interested in whether or not C caused E. By Con we denote the quadruple (T, b, C, E), and refer to this as a *context*.

3.1 Actual Causation in General

For reasons of simplicity, the majority of approaches (including Hall) only consider actual causation in a deterministic setting. Further, it is taken for granted that the actual values of all variables are given. In such a context, counterfactual dependence of the event E on C is expressed by the conditional: *if* $do(\neg C)$ *then* $\neg E$, where it is assumed that all exogenous variables take on their actual values. In our probabilistic setting, the latter translates into making those laws that were actually applied deterministic, in accordance with the choices made in the story. However in many cases some exogenous variables simply do not have an actual value to start with. For example, if Suzy is prevented from throwing her rock, then we cannot say what the accuracy would have been had she done so. In CP-logic, this would be represented by the fact that law (3) is not applied. Hence, in a more general setting, it is required only that $do(\neg C)$ makes

$\neg E$ possible. In other words, we get a probabilistic definition of counterfactual dependence:

Definition 1 (Dependence). *E is* counterfactually dependent *on C in (T, b) iff*

$P_{T^b}(\neg E | do(\neg C)) > 0$.

As counterfactual dependency lies at the heart of causation for all of the approaches we are considering, Dependence represents the most straightforward definition of actual causation. It is however too crude and allows for many counterexamples, preemption being the most famous.

More refined definitions agree with the general structure of the former, but modify the theory T in more subtle ways than T^b does. We identify two different kinds of laws in T, that should each be treated in a specific way. The first are the laws that are *intrinsic* with respect to the given context. These should be made deterministic in accordance with b. The second are laws that are *irrelevant* in the given context. These should simply be ignored. Together, the methods of determining which laws are intrinsic and irrelevant, respectively, will be the parameters of our general definition. Suppose we are given two functions Int and Irr, which both map each context (T, b, C, E) to a subset of the theory T. With these, we define actual causation as follows:

Definition 2 (Actual causation given Int and Irr). *Given the context Con, we define that C is an actual cause of E if and only if E is counterfactually dependent on C according to the theory T' that we construct as:*

$T' = [T \setminus (Irr(Con) \cup Int(Con))] \cup Int(Con)^b$.

For instance, the naive approach that identifies actual causation with counterfactual dependence corresponds to taking Irr as the constant function $\{\}$ and $Int(Con)$ as $\{r \in T \mid r \text{ was applied in } b\}$. From now on, we use the following, more legible notation for a particular instantiation of this definition:

Irr-Dependence 1 *No law r is irrelevant.*

Intr-Dependence 1 *A law r is intrinsic iff r was applied in b.*

If desired, we can order different causes by their respective counterfactual probabilities, as this indicates how important the cause was for E.

3.2 Beckers and Vennekens 2012 Definition

A recent proposal by the current authors for a definition of actual causation was originally formulated in [13], and later slightly modified in [1]. Here, we summarize the basic ideas of the latter, and refer to it as *BV12*. We reformulate this definition in order to fit into our framework. It is easily verified that both versions are equivalent.

Because we want to follow the actual story as closely as possible, the condition for intrinsicness is exactly like before: we force all laws that were applied in b to have the same effect as they had in b.

To decide which laws were relevant for causing E in our story, we start from a simple temporal criterion: every law that was applied after the effect E took place is irrelevant, and every law that was applied before isn't. For example, to figure out why the bottle broke in our previous example, law (4) is considered irrelevant, because the bottle was already broken by the time Billy's rock arrived. For laws that were not applied in b, we distinguish laws that could still be applied when E occurred, from those that could not. The first are considered irrelevant, whereas the second aren't. This ensures that any story b' that is identical to b up to and including the occurrence of E provides the same judgements about the causes of E, since any law that is not applied in b but is applied in b', must obviously occur after E.

Irr-BV12 1 *A law r is irrelevant iff r was not applied before E in b, although it could have. (I.e., it was not impossible at the time when E occurred.)*

Intr-BV12 1 *A law r is intrinsic iff r was applied in b.*

3.3 Hall 2007

One of the currently most refined concepts of actual causation is that of [4]. Although Hall uses structural equations as a practical tool, he is of the opinion that intuitions about actual causation are best illustrated using neuron diagrams. A key advantage of these diagrams, which they share with CP-logic, is that they distinguish between a default and deviant state of a variable. Proponents of structural equations, on the other hand, countered Hall's approach by criticizing neuron diagrams' limited expressivity [9, p. 398]. Indeed, a neuron diagram, and thus Hall's approach as well, is very limited in the kind of examples it can express. In particular, neuron diagrams can only express deterministic causal relations and they also lack the ability to directly express *causation by omission*, i.e., that the absence of C causes E. Hall's solution is to argue against causation by omission altogether. By contrast, we will offer an improvement of Hall's account that generalizes to a probabilistic context, and can also handle causation by omission. In short, we propose CP-logic as a way of overcoming the shortcomings of both structural equations and neuron diagrams.

In a neuron diagram, a neuron can be in one of two states, the default "off" state and the deviant "on" state in which the neuron "fires". Different kinds of links define how the state of one node affects the other. For instance, in (a), E fires iff at least one of B or D fires, D fires iff C fires, and B fires iff A fires and C doesn't fire. Nodes that are "on" are represented by full circles and nodes that are "off" are shown as empty circles.

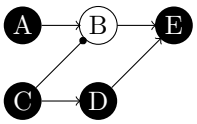


Fig. 2: (a)

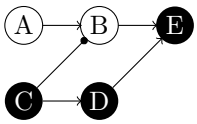


Fig. 3: (b)

Diagrams (a) and (b) represent the same causal structure, but different stories: in both cases there are two causal chains leading to E, one starting with C and another starting with A. But in (a) the chain through B is preempted by C, whereas in (b) there is nothing for C to preempt, as A doesn't even fire. Therefore (a) is an example of what is generally known as Early Preemption, whereas (b) is not.

Although Hall presents his arguments using neuron diagrams, his definitions are formulated in terms of structural equations that correspond to such diagrams in a straightforward way: for each endogenous variable there is one equation, which contains a propositional formula on the right concisely expressing the dependencies of the diagram.

Any structural model M can also be read as a CP-logic theory T. The firing of the neurons and the resulting assignment to the variables in M, then correspond to a story b.

One important difference between structural equations and CP-laws, is that we are not limited to using a single CP-law for each variable. As each law represents a separate causal mechanism, and only one mechanism can actually make a variable become **true**, we will represent dependencies such as that of E by three laws, corresponding to the three different ways in which B and D can cause E: each by themselves, or the two of them simultaneously. At first sight the conjunctive law may seem redundant, but if one has a temporal condition for irrelevance – eg. BV12 – then it may not be. The translation of examples (a) and (b) into CP-logic is given by the following CP-theory – where p and q represent some probabilities:

$$\begin{array}{lll} (A:p) \leftarrow . & B \leftarrow A \wedge \neg C. & E \leftarrow B. \\ (C:q) \leftarrow . & D \leftarrow C. & E \leftarrow D. \\ & & E \leftarrow B \wedge D. \end{array}$$

The idea behind Hall's definition is to check for counterfactual dependence in situations which are reductions of the actual situation, where a reduction is understood as "a variant of this situation in which *strictly fewer* events occur". In other words, because the counterfactual dependence of E on C can be masked by the occurrence of events which are extrinsic to the actual causal process, we look at all possible scenario's in which there are less of these extrinsic events. Hall puts it like this [4, p. 129]:

> Suppose we have a causal model for some situation. The model consists of some equations, plus a specification of the actual values of the variables. Those values tell us how the situation *actually* unfolds. But the same system of equations can also represent *nomologically possible variants*:

> just change the values of one or more exogenous variables, and update the rest in accordance with the equations. A good model will thus be able to represent a range of variations on the actual situation. Some of these variations will be – or more accurately, will be modeled as – *reductions* of the actual situation, in that every variable will either have its actual value or its default value. Suppose the model has variables for events C and E. Consider the conditional
>
> $$\textbf{if } \boldsymbol{C = 0}\textbf{; then } \boldsymbol{E = 0}$$
>
> This conditional may be true; if so, C is a cause of E. Suppose instead that it is false. Then C is a cause of E iff there is a reduction of the actual situation according to which C and E still occur, and in which this conditional is true.

Rather than speaking of fewer events occuring, in this definition Hall characterizes a reduction in terms of whether or not variables retain their actual value. This is because in the context of neuron diagrams, an event is the firing of a neuron, which is represented by a variable taking on its deviant value, i.e., the variable *becoming true*. In the dynamic context of CP-logic, the formal object that corresponds most naturally to Hall's informal concept of an event is the transition in a probability tree (i.e., the application of a causal law) that makes such a variable true. Therefore we take a reduction to mean that no law is applied such that it makes a variable true that did not become true in the actual setting.

To make this more precise, we introduce some new formal terminology. Let d be a branch of a probability tree of the theory T. $Laws_d$ denotes the set of all laws that were applied in d. The resulting effect of the application of a law $r \in Laws_d$ – i.e., the disjunct of the head which was chosen – will be denoted by r_d, or by 0 if an empty disjunct was chosen. The set of true variables in the leaf of d will be denoted by $Leaf_d$.

A branch d is a *reduction* of b iff $\forall r \in Laws_d : r_d = 0 \vee \exists s \in Laws_b : r_d = s_b$. Or, equivalently, $Leaf_d \subseteq Leaf_b$.

A reduction of b in which both C and E occur – i.e., hold in its leaf – will be called a (C, E)-reduction. The set of all of these will be denoted by $Red_b^{(C,E)}$. These are precisely the branches which are relevant for Hall's definition.

Definition 3. *We define that C is an* actual cause *of E iff*
$(\exists d \in Red_b^{(C,E)} : P_{T^d}(\neg E | do(\neg C)) > 0)$.

Theorem 1 shows the correctness of our translation. Proofs of all theorems can be found in the Appendices.

Theorem 1. *Given a neuron diagram with its corresponding equations M, and an assignment to its variables V. Consider the CP-logic theory T and story b that we get when applying the translation discussed above. Then C is an actual cause of E in the diagram according to Hall's definition iff C is an actual cause of E in b and T according to Definition 3.*

At first sight, Definition 3 does not fit into the general framework we introduced earlier, because of the quantifier over different branches. However, we will now show that for a significant group of cases it actually suffices to consider just a single T', which can be described in terms of irrelevant and intrinsic laws.

Rather than looking at all of the reductions separately, we single out a minimal structure which contains the essence of our story. In general such a minimal structure need not be unique, as the story may contain elements none of which are necessary by themselves yet without all of them the essence is changed. The following makes this more precise.

Definition 4. *A law r is* necessary *iff*

- $\forall d \in Red_b^{(C,E)} : r \in Laws_d$ *and*
- $\forall d, e \in Red_b^{(C,E)} : r_d = r_e$.

We define $Nec(b)$ as the set of all necessary laws.

In general it might be that there are two (or more) edges which are unnecessary by themselves, but at least one of them has to be present. Consider for example a case where C causes both A and B, and each of those in return is sufficient to cause E. Then neither the law $r = A : ... \leftarrow C$ nor the law $r' = B : ... \leftarrow C$ is necessary, yet at least one of them has to be applied to get E. In cases where this complication does not arise, we shall say that the story is simple.

Definition 5. *A story b is* simple *iff the following holds:*

- $\forall r \in Laws_b$: *the head of r contains at most two disjuncts;*
- $\forall d \in Red_b^{(C,E)}$, *for all non-deterministic $r \in Laws_d \setminus Nec(b) : \exists e \in Red_b^{(C,E)}$ so that $e = d$ up to the application of r, and $r_d \neq r_e$.*

As an example, note that the story in the previous paragraph is not simple. Neither law r nor r' is necessary. Now consider the (C, E)-reduction d where first r' fails to cause B, followed by r causing A, which in turn causes E. The branch that is identical to d up to and including the application of r' but in which r does not cause A, is not a (C, E)-reduction.

We are now in a position to formulate a theorem that will allow us to adjust Hall's definition into our framework.

Theorem 2. *If $(\exists d \in Red_b^{(C,E)} : P_{T^d}(\neg E|do(\neg C)) > 0)$ then $P_{T^{Nec(b)}}(\neg E|do(\neg C)) > 0$. If b is simple, then the reverse implication holds as well.*

It is possible to add an additional criterion to turn this theorem into an equivalence that also holds for non-simple stories. We choose not to do this, because all of the examples Hall discusses are simple, as are all of the classical examples discussed in the literarure, such as Early and Late Preemption, Symmetric Overdetermination, Switches, etc.

As a result of this theorem, rather than having to look at all (C, E)-reductions and calculate their associated probabilities, we need only find all the necessary

laws and calculate a single probability. If the story b is simple, then this probability represents an extension of Hall's definition, since they are equivalent if one ignores the value of the probability but for it being 0 or not. To obtain a workable definition of actual causation, we present a more constructive description of necessary laws. From now on we call the node resulting from the application of a law r in b $Node_r^b$.

Theorem 3. *If b is simple, then a non-deterministic law r is necessary iff there is no (C, E)-reduction passing through a sibling of $Node_r^b$.*

With this result, we can finally formulate our version of Hall's definition, which we will refer to as Hall07.

Irr-Hall07 1 *No law r is irrelevant.*

Intr-Hall07 1 *A law r is intrinsic iff r was applied in b, and there is no branch d passing through a sibling of $Node_r^b$ such that $\{C, E\} \subseteq Leaf_d \subseteq Leaf_b$.*

3.4 Hall 2004 Definitions

[3] claims that it is impossible to account for the wide variety of examples in which we intuitively judge there to be actual causation by using a single, all-encompassing definition. Therefore he defines two different concepts which both deserve to be called forms of causation but are nonetheless not co-extensive.

Dependence The first of these is simply Dependence, as stated in Definition 1. As mentioned earlier, Hall only considers deterministic causal relations, and thus the probabilistic counterfactual will either be 1 or 0.

Production The second concept tries to express the idea that to cause something is to bring it about, or to *produce* it. The original, rather technical, definition can be found in the appendices, but the following informal version suffices for our purposes: C is a producer of E iff there is a directed path of firing neurons in the diagram from C to E. In our framework, this translates to the following.

Irr-Production 1 *A law r is irrelevant iff r was not applied before E in b, or if its effect was already* **true** *when it was applied.*

Intr-Production 1 *A law r is intrinsic iff r was applied in b.*

Theorem 4. *Given a neuron diagram with its corresponding equations M, and an assignment to its variables V. Consider the CP-logic theory T, and a story b, that we get when applying the translation discussed earlier. C is a producer of E in the diagram according to Hall iff C is a producer of E in b and T according to the CP-logic version stated here.*

Besides providing a probabilistic extension, the CP-logic version of production also offers a way to make sense of causation by omission. That is, just as with all of the definitions in our framework in fact, we can extend it to allow negative literals such as $\neg C$ to be causes as well.

4 Comparison

Table 1 presents a schematic overview of the four definitions discussed so far, as well as two new ones, that we give appropriate names. The columns and rows give the criteria for a law r of T to be considered intrinsic, respectively irrelevant, in relation to a story b, and an event E. By $r \leq_b E$, we denote that r was applied in b before E occurred.

Table 1: Spectrum of definitions

Irrelevant	Intrinsic	
	$r \in Laws_b$	$r \in Nec(b)$
$\emptyset$	Dependence	Hall07
$\exists d : (d = b \text{ up to } E) \wedge r \geq_d E$	BV12	BV07
$r \not\leq_b E \vee r_b <_b r$	Production	Production07

In order to illustrate the working of the definitions and to highlight their differences, we present an example:

> *Assassin* decides to poison the meal of a victim, who subsequently *Dies* right before dessert. However, $Murderer$ decided to murder the victim as well, so he poisoned the dessert. If *Assassin* had failed to do his job, then *Backup* would have done so all the same.

The causal laws that form the context of this story are give by the following theory:

$$(Assassin : p) \leftarrow . \qquad Dies \leftarrow Assassin.$$
$$(Murderer : q) \leftarrow . \qquad Dies \leftarrow Backup.$$
$$(Backup : r) \leftarrow \neg Assassin. \qquad Dies \leftarrow Murderer.$$

In this story, did *Assassin* cause *Dies*? We leave it to the reader to verify that in this case the left intrinsicness condition from the table applies to the first two non-deterministic laws, whereas the right one only applies to the first. The second irrelevance condition only applies to the last law, whereas the third one applies to the last two laws and to the third. This results in the following probabilities representing the causal status of *Assassin*:

Production	BV12	Hall07	Dependence
1	$1 - r$	$(1 - r) * (1 - q)$	0

Different motivations can be provided for these answers:

– **Production**: *Assassin* brought about the death of the victim all by himself, hence he is the full cause.

- **BV12**: If *Assassin* hadn't killed him, then that omission itself would not have lead to victim's death with a probability of $(1-r)$. Hence, *Assassin* is a cause of the death to this extent.
- **Hall07**: Ignoring the actually redundant $Murderer$, if *Assassin* doesn't kill him, then there is a $(1-r)*(1-q)$ probability that the victim will die. Hence he is the cause to that extent.
- **Dependence**: The victim would have died anyway, so *Assassin* is not a cause at all.

Rather than saying that only one of these answers is correct, we prefer to think of them as answering different questions, all of which have their use in some context or other. (Eg., to determine responsibility, understand Assassin's state of mind, minimize the chance of murders, etc.) More generally, the definitions could be characterized by describing which events are allowed to happen in the counterfactual worlds they take into consideration to judge causation.

- **Production**: Only those events – i.e., applications of laws – which led to E, and not differently – i.e., with the same outcome as in the actual story.
- **BV12**: Those events which led to E, and not differently, and also those events which were prevented from happening by these.
- **Hall07**: Any event can happen, as long as those events that were essential to lead to E do not happen differently.
- **Dependence**: Any event can happen, as long as those events that did actually happen do not happen differently.

5 Conclusion

In this paper we have used the formal language of CP-logic to formulate a general definition of actual causation, which we used to express four specific definitions: a proposal of our own, and three definitions based on the work of Hall. By moving from the deterministic context of neuron diagrams to the non-deterministic context of CP-logic, the latter definitions improve on the original ones in two ways: they can deal with a wider class of examples, and they allow for a graded judgment of actual causation in the form of a conditional probability. Further, comparison between the definitions is facilitated by presenting them as various ways of filling in two central concepts. We have illustrated the flexibility of CP-logic in expressing different definitions, opening the path to other proposals beyond the ones here discussed.

6 Appendices

To facilitate the proof of the first theorem, we introduce the following lemma.

Lemma 1. *Given a neuron diagram D with its corresponding equations M, and an assignment to its variables V. Consider the CP-logic theory T, and a story*

b, that we get when applying the translation discussed earlier. Then a neuron diagram R is a reduction of D in which both C and E occur iff its translation d – another branch of T – is a (C, E)-reduction of b.

Proof. Assume we have a reduction R of a neuron diagram D, and b is the story corresponding to D. As R is simply a different assignment of the variables occurring in D, brought about by the same equations that existed for D, this reduction corresponds to another branch d of T, in which C and E hold in its leaf. Moreover, R can be constructed starting from D by changing some of the exogenous variables, say U', from their actual values to their default value, and then updating the endogenous variables in accordance with the deterministic equations. It being a reduction, this caused no new variables to take on their deviant value in comparison to D. Let r be a law that occurs in d.

If r is non-deterministic, it must be one of the laws representing an exogenous variable V, i.e., a law with an empty body, and hence it was also applied in b. R being a reduction, either V has the same value in R as in the original diagram, or it has its default value. In the former case, this means that $r_d = r_b$, in the latter case $r_d = 0$, both of which satisfy the requirement for d being a reduction.

If r is deterministic, the precondition for r has to be fulfilled in d, causing some variable V to take on its deviant value. The same must hold true of the precondition for the equation for V, and thus V takes on its deviant value in R as well, implying it did so in D too. Therefore there must have been some law applied in b that made V take on its deviant value as well. From this it follows that d is a (C, E)-reduction of b.

Now assume we have a theory T and a story b that form the translation of a neuron diagram D, such that C and E hold in b, and that d is a (C, E)-reduction of b. As the leaf of d contains an assignment to all of the variables that satisfies the equations of M, there is a neuron diagram R that corresponds to d. We can easily go over all the previous steps in the other direction, to conclude that R is a reduction of D in which C and E are true.

Theorem 1. *Given a neuron diagram with its corresponding equations M, and an assignment to its variables V. Consider the CP-logic theory T and story b that we get when applying the translation discussed above. Then C is an actual cause of E in the diagram according to Hall's definition iff C is an actual cause of E in b and T according to Definition 3.*

Proof. We start with the implication from left to right. Assume we have a neuron diagram D, in which both C and E fire. This translates into a theory T and a story b, for which C and E hold in its leaf. Further, assume there is a reduction R of this diagram, in which both C and E continue to hold, and in this reduction, if $C = 0$; then $E = 0$. By the above lemma, this translates into a (C, E)-reduction of b, say d.

In R, if $C = 0$; then $E = 0$. The conditional $C = 0$ is interpreted as a counterfactual locution, and corresponds to $do(\neg C)$. As there are no non-deterministic laws with non-empty preconditions, T^d is simply the deterministic theory that

determines the same assignment as R, meaning $P_{T^d}(\neg E|do(\neg C)) = 1$, which concludes this part of the proof.

Now assume we have a theory T and a story b that form the translation of a neuron diagram D, such that C and E hold in b, and that d is a (C, E)-reduction of b for which the given inequality holds. By the above lemma, the translation of d, say R, is reduction of D in which C and E occur. As mentioned in the previous paragraph, T^d simply corresponds to an assignment of values to the variables occurring in D that follows its equations. Since R describes this same assignment, in R too if $C = 0$; then $E = 0$. This concludes the proof.

Theorem 2. *If* $(\exists d \in Red_b^{(C,E)} : P_{T^d}(\neg E|do(\neg C)) > 0)$ *then* $P_{T^{Nec(b)}}(\neg E|do(\neg C)) > 0$. *If* b *is simple, then the reverse implication holds as well.*

Proof. We start with proving the first implication. Assume we have a $d \in Red_b^{(C,E)}$ such that $P_{T^d}(\neg E|do(\neg C)) > 0$. This implies that there is at least one branch e of a probability tree of $T^d|do(\neg C)$ for which $\neg E$ holds in its leaf. We prove by induction on the length of e that this implies the existence of a similar branch e' of a probability tree of $T^{Nec(b)}|do(\neg C)$ for which $\neg E$ holds in its leaf, which is what is required to establish the theorem.

Base case: if e consists of a single node – i.e., the root node where all atoms are false – then this means that no laws of $T^d|do(\neg C)$ can be applied. Since the bodies of the laws in $T^{Nec(b)}|do(\neg C)$ are identical to those of the laws in $T^d|do(\neg C)$, we simply have $e' = e$.

Induction case: Assume we have a sub-branch e_n of e with length $n > 1$, starting from the root node, and that we also have a structurally identical sub-branch e'_n. By it being structurally identical we mean that they are identical except for the fact that they may have different probabilities along the edges.

If $e_n = e$, then no more laws can be applied in the final node of e_n. This must then hold for the final node of e'_n as well, so we are finished. Otherwise, we know that there is a sub-branch e_{n+1} which extends e_n along e with a node O. Assume that the law which was applied to get to O is r.

If r is deterministic, then r occurs in $T^d|do(\neg C)$ exactly as it does in $T^{Nec(b)}|do(\neg C)$. Since both branches are structurally identical, e'_n can be extended in the exact same manner as e_n, so there has to be a probability tree of $T^{Nec(b)}|do(\neg C)$ in which there is a sub-branch e'_{n+1} with the desired properties. So assume r is non-deterministic.

First assume $r \notin Laws_d$. This implies that $r \notin Nec(b)$. So as in the deterministic case, r occurs in $T^d|do(\neg C)$ exactly as it does in $T^{Nec(b)}|do(\neg C)$, and the branch can be extended in the same manner.

Now assume $r \in Laws_d$. If also $r \in Nec(b)$, we know that $r_d = r_b = r_{Nec}$ and hence the previous argument holds. Remains the possibility that $r \notin Nec(b)$. As in the deterministic case, because r can be applied in the final node of e'_n there has to be a probability tree of $T^{Nec(b)}|do(\neg C)$ with a sub-branch like e'_n where r is applied next.

Assume $r_d = A$. Since A was the outcome of r in d, the law r as it appears in T – and also in $T^{Nec(b)}|do(\neg C)$ – contains A in its head with some probability

attached to it. Therefore the final node of e'_n in the said probability tree has one child-node which contains A, extending e'_n into a sub-branch e'_{n+1} with the desired properties. This concludes this part of the proof.

Now we prove that if b is simple, the reverse implication holds as well.

Assume $P_{T^{Nec(b)}}(\neg E|do(\neg C)) > 0$. This implies that there is at least one branch e of a probability tree of $T^{Nec(b)}|do(\neg C)$ for which $\neg E$ holds in its leaf. We can repeat the first steps of the previous implication, so that we again arrive at a law r which was applied to get to a node O.

The branch e' we are considering occurs in a probability tree of a (C, E)-reduction, say f. First assume $r \in Nec(b)$. By definition, this implies that also $r \in Laws_f \wedge r_{Nec} = r_f$, and we can apply the reasoning from above. Likewise as above, we can apply this reasoning to all other cases, except the one where $r \notin Nec(b)$, r is non-deterministic, and $r \in Laws_f$. Assume the law r has effect A in the branch e we are considering. If $r_f = A$, then we are back to our familiar situation, so therefore assume $r_f = B$, and $A \neq B$.

Since b is simple, A and B are the only two possible effects of r. Further, remark that $r \in Laws_b \setminus Nec(b)$. This implies the existence of a (C, E)-reduction g that is identical to f up to the application of r, but such that $r_g \neq r_f$, and thus $r_g = A = r_e$ meaning there is a branch in a probability tree of g that is structurally identical to e up to O. This concludes the proof of the theorem.

Theorem 3. *If b is simple, then a non-deterministic law r is necessary iff there is no (C, E)-reduction passing through a sibling of $Node_r^b$.*

Proof. Say the unique sibling of $Node_r^b$ is M. We start with the implication from left to right, so we assume r is necessary. Assume $r_b = A$, then there is no $d \in Red_b^{(C,E)}$ for which $r_d \neq A$, hence there is no (C, E)-reduction which passes through M.

Remains the implication from right to left. Assume we have a law r such that there is no (C, E)-reduction passing through a sibling of $Node_r^b$. We proceed with a reductio ad absurdum, so we assume r is not necessary.

Clearly b is a (C, E)-reduction of itself, and also $r \in Laws_b \setminus Nec(b)$. Hence, by b's simplicity, there is a (C, E)-reduction e which is identical to b up to the application of r, but for which $r_e \neq r_b$. Thus e passes through the sibling of $Node_r^b$, contradicting the assumption that r is necessary. This concludes the proof.

Theorem 4. *Given a neuron diagram with its corresponding equations M, and an assignment to its variables V. Consider the CP-logic theory T, and a story b, that we get when applying the translation discussed earlier. C is a producer of E in the diagram according to Hall iff C is a producer of E in b and T according to the CP-logic version stated here.*

Proof. First we need to explain some terminology that Hall uses. A *structure* is a temporal sequence of sets of events, which unfold according to the equations of some neuron diagram. A branch, or a sub-branch, would be the corresponding concept in CP-logic.

Two structures are said to *match intrinsically* when they are represented in an identical manner. The reason why Hall uses this term, is because even though we use the same variable for an event occurring in different circumstances, strictly speaking they are not the same. This is mainly an ontological issue, which need not detain us for our present purposes.

A set of events S is said to be *sufficient* for another event E, if the fact that E occurs follows from the causal laws, together with the premisse that S occurs at some time t, and no other events occur at this time. A set is *minimally* sufficient if it is sufficient, and no proper subset is. To understand this, note that the ambiguity of the relation between an event and the value of a variable that we noted earlier, resurfaces here. In the context of neuron diagrams, events are temporal, and occur during the time-period that a neuron fires, i.e, becomes true. However, at any later time-point, the variable corresponding to this neuron will remain to be true, implying that the value of the variable has shifted in meaning from "the neuron fires" to "the neuron has fired". Given this interpretation, it is natural to translate Hall's notion of an event into CP-logic as the application of a law, making a variable true, as we have done.

A further detail to be cleared out, is that in the context of neuron diagrams there can be simultaneous events, since multiple neurons can fire at the same time. In CP-logic, in each node only one law is allowed to be applied, hence this translates to two consecutive edges in a branch. Therefore it is not the case that each node-edge pair in a branch corresponds to a separate time-point, but rather sets of consecutive pairs – with variable size – do. Given such a set, then for each variable that was the result of the application of a law belonging to it, it holds that its corresponding event occurs at the next time-point, corresponding to the next set of nodes further down the branch. All the variables occuring in the bodies of the laws in this set, represent events that occur during this time-point.

Now we can state the precise definition of production as it occurs in [3, p.25].

> We begin as before, by supposing that E occurs at t', and that t is an earlier time such that at each time between t and t', there is a unique minimally sufficient set for E. But now we add the requirement that whenever t_0 and t_1 are two such times ($t_0 < t_1$) and S_0 and S_1 the corresponding minimally sufficient sets, then
>
> – for each element of S_1, there is at t_0 a unique minimally sufficient set; and
> – the union of these minimally sufficient sets is S_0.
>
> ...
>
> Given some event E occurring at time t' and given some earlier time t, we will say that E has a *p*ure causal history back to time t just in case there is, at every time between t and t', a unique minimally sufficient set for E, and the collection of these sets meets the two foregoing constraints. We will call the structure consisting of the members of these sets the "pure causal history" of E, back to time t. We will say that C is a proximate cause of E just in case C and E belong to some structure

> of events S for which there is at least one nomologically possible structure S' such that (i) S' intrinsically matches S; and (ii) S' consists of an E-duplicate, together with a pure causal history of this E-duplicate back to some earlier time. (In easy cases, S will itself be the needed duplicate structure.) Production, finally, is defined as the ancestral [i.e., the transitive closure] of proximate causation.

We will start with the implication from left to right. So assume we have a neuron diagram D, in which C is a producer of E. Say T is the CP-logic theory that is the translation of the equations of the diagram, and b is the branch representing the story. We already know that C and E hold in the leaf of b. We need to proof that $P_{T'}(\neg E|do(\neg C)) > 0$. The theory T' only contains deterministic laws, and no disjunctions, hence all its laws are of the form: $V \leftarrow A \wedge A' \wedge ... \wedge \neg B \wedge \neg B'$, where the number of positive literals in the conjunction is at least one. Therefore any probability tree for T' consists out of only one branch, determining a unique assignment for all the variables. Further, even though the theory T may contain several laws in which a variable occurs in the head, because of our irrelevance criterion T' contains exactly one law for every variable that is true. So for every true variable in this assignment, there is a unique chain of laws – neglecting the order – which needs to be applied to make this variable true. For any such variable V, we will say that it depends on all of the variables occurring positively in the body of a law in this chain. Clearly, if any true variable changes its value in this assignment, then all variables which depend on it become false.

As a first case, assume C is a proximate cause of E. We start by assuming that circumstances are nice, meaning that D contains itself a structure S which is a pure causal history of E. This means that in the actual story b, C is part of a unique minimally sufficient set for E. From this it follows that in T', C figures positively in one of the laws on which E depends. Hence, if we apply $do(\neg C)$, then E will no longer hold.

Now assume that there is a structure S occurring in D, such that there exists another diagram, say D', in which this structure occurs as well, and forms a pure causal history of E. This diagram corresponds to a branch of T, say d, That means that in T'_d – i.e., the theory T' constructed out d – C occurs positively in the unique chain of laws which can make E true. But as all events in S also occur in D, at the same moments as they do in D', that means that C must also occur positively in the unique chain of laws for E in the theory T'_b. Hence, E depends on C in the theory T'_b as well.

Now look at the more general case, in which C occurs in a chain of proximate causes, that leads up to E. I.e, in D, C is the proximate cause of some variable V_1, which in turn is the proximate cause of some variable V_2, and so on until we get to E. We know from the previous discussion, that this implies in T' that $do(\neg C)$ then $\neg V_1$, and $do(\neg V_1)$ then $\neg V_2$, and so on. Given what we know about T', it directly follows that when we apply $do(\neg C)$, then $\neg E$. This concludes this part of the proof.

We continue with the implication from right to left. So assume that we are given again a neuron diagram and a corresponding story b, and that we know $P_{T'}(\neg E|do(\neg C)) > 0$. From our earlier analysis of T', we know that this means that C occurs positively in the unique chain of laws that can make E true according to T'. From this chain of laws, we start from the one causing E and from there pick out a series that gets us to a law where C occurs positively in the body. More concretely, we take a series of the form: $E \leftarrow ...A \wedge ...$, $A \leftarrow ...D \wedge ...$, and so on until we get at a law $Z \leftarrow ...C \wedge ....$ By definition of production, it suffices to prove that in this chain, each of the variables in the body is a proximate cause of the variable in the head.

Take such a law $V \leftarrow ...W \wedge ....$ At the time that this law is applied, W clearly is a member of a sufficient set of events for V, which occurs at the next time point. Say S_0 is the set of all events that occur together with W that figure in the body of this law, and S_1 is the set $\{V\}$ that occurs at the next time-point, then the structure consisting precisely of S_0 and S_1 and nothing else forms a pure causal history of V containing W. The same reasoning applies to all laws of the chain. This concludes the proof.

References

1. Beckers, S. and Vennekens, J. 2012. Counterfactual dependency and actual causation in CP-logic and structural models: a comparison. *STAIRS 2012: Proceedings of the Sixth Starting AI Researchers Symposium*, 35-46.
2. Hall, N. and Paul, L.A. 2003. Causation and preemption. *Philosophy of Science Today.* Oxford University Press.
3. Hall, N. 2004. Two concepts of causation. In *Causation and Counterfactuals.*
4. Hall, N. 2007. Structural equations and causation. *Philosophical Studies 132,* 1, 109–136.
5. Halpern, J. and Pearl, J. 2005. Causes and explanations: A structural-model approach. part I: Causes. *The British Journal for the Philosophy of Science 56,* 4, 843–87.
6. Halpern, J. and Pearl, J. 2005. Causes and explanations: A structural-model approach. part II: Explanations. *The British Journal for the Philosophy of Science 56,* 4.
7. Halpern, J. and Hitchcock, C. 2015. Graded Causation and defaults. *The British Journal for the Philosophy of Science 66,* 2, 413–457.
8. Hitchcock, C. 2007. Prevention, preemption, and the principle of sufficient reason. *Philosophical review 116,* 4, 495–532.
9. Hitchcock, C. 2009. Structural equations and causation: six counterexamples. *Philosophical Studies 144*, 391–401.
10. Lewis, D. 1973. Causation. *J. of Philosophy 70*, 113–126.
11. Pearl, J. 2000. *Causality: Models, Reasoning, and Inference.* Cambridge University Press.
12. Shafer, G. 1996. *The art of causal conjecture.* MIT Press.
13. Vennekens, J. 2011. Actual causation in CP-logic. *Theory and Practice of Logic Programming 11*, 647–662.
14. Vennekens, J., Denecker, M., and Bruynooghe, M. 2009. CP-logic: A language of probabilistic causal laws and its relation to logic programming. *Theory and Practice of Logic Programming 9*, 245–308.

15. Vennekens, J., Denecker, M., and Bruynooghe, M. 2010. Embracing events in causal modelling: Interventions and counterfactuals in CP-logic. In *JELIA*. 313–325.

Most Probable Explanation for MetaProbLog and its application in Heart Sound Segmentation

Theofrastos Mantadelis[1], Jorge Oliveira[2] and Miguel Coimbra[2]

{CRACS & INESC TEC[1]; Instituto de Telecomunicações[2]},
Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre 1021/1055,
4169-007 Porto, Portugal
{theo.mantadelis; oliveira_jorge; mcoimbra}@dcc.fc.up.pt

Abstract. This paper, presents ongoing work that extends MetaProbLog with Most Probable Explanation (MPE) inference method. The MPE inference method is widely used in Hidden Markov Models in order to derive the most likely states of a model. Recently, we started developing an application that uses MetaProbLog to models phonocardiograms. We target to use this application in order to diagnose heart diseases by using phonocardiogram classification. Motivated by the importance of phonocardiogram classification, we started the implementation of the MPE inference method and an improvement of representation for annotated disjunctions.

1 Introduction

MetaProbLog[1] is a framework of the ProbLog [4,6] probabilistic logic programming language. ProbLog extent Prolog programs by annotating facts with probabilities. In that way it defines a probability distribution over all Prolog programs. ProbLog follows the distribution semantics presented by Sato [12]. MetaProbLog first appeared in [10] where the focus was to solve meta calls for ProbLog.

The goal of this paper is to present ongoing work which extends MetaProbLog with MPE inference. In Hidden Markov Models (HMM) this inference task is known as the most likely sequence of hidden states and it is been solved with the Viterbi algorithm [15]. Similarly, in Bayesian Networks this inference task is called Maximum a posteriori (MAP) and is usually been estimated either by Monte Carlo approximation or by an expectation maximization (EM) algorithm.

ProbLog semantics allow the description of different type of models, including HMM, Bayesian Networks, and others. Solving this inference task for ProbLog programs, boils down to finding the most probable witness of satisfiability. The original implementation of ProbLog[2] used a greedy search with pruning to find the MPE [7], this algorithm was called MAX. While for the initial set of ProbLog features the MAX algorithm was sufficient, newly added features have increased

[1] https://www.dcc.fc.up.pt/metaproblog/tiki-index.php?page=HomePage
[2] https://dtai.cs.kuleuven.be/problog/problog1/problog1.html

the challenge for finding the MPE. ProbLog 2[3] [5] computes the MPE by first, collecting all explanations as an AND-OR tree then by handling the introduced cycles and finally traversing the tree to find the MPE. This approach currently is the most complete and sound approach but does not take advantage of any pruning strategy.

These new features are crucial for the implementation of an application that classifies phonocardiogram (PCG) signals. We intent to use MetaProbLog in order to implement an application that identifies the most likely characterizations of PCG signals. These characterized PCG signals can then be used in classification to diagnose heart diseases.

2 A Motivating Application

Lately, the classification of PCG signals has got significant attention in the academic community [1]. Classifying PCGs is both a challenging and an important task. Heart sounds are non-trivial signals, since they might contain non-stationary noise, have artifacts and murmur sounds. Heart sound auscultation techniques is one of the most reliable and successful tools in early diagnosis used for potentially deadly heart diseases, such as natural and prosthetic heart valve dysfunction or even in heart failure. Therefore a computer-aided auscultation may allow detection of diseases that are hardly recognized through the traditional methods, for instance ischemic heart disease.

Several factors may complicate S1 (first heart sound) and S2 (second heart sound) detection, such as variability in heart rhythm, dynamic background noise, anatomical variations, artifacts, murmurs, respiratory sounds interference, clicks, and extra-sounds such as S3 and S4 sounds. These factors in combination, result in a low signal-to-noise ratio. A first fundamental step for the PCG analysis is to segment the signal into periods. Several algorithms were successfully implemented in heart sound segmentation problem, such as S-transform [11], recognition system based on Neural Networks [17] and Wavelet Decomposition [2].

Most of the heart sounds used in heart sound segmentation are primarily recorded with specialized equipment and in a controlled environment, ensuring a very high signal-to-noise ratio. However, routine sounds that are obtained with handheld stethoscopes in clinical environments (such as screening campaigns like Caravana do Coracão[4] from where we possess data) have a low signal-to-noise ratio. The correct identification of heart sounds is crucial for the analysis of these signals in more detail.

Recently, HMMs have being used for modeling and characterizing real-word signals such as heart sound signals [13]. We aim to model PCG signals as a HMM and use MetaProbLog to find the most likely sequence of events (S1, S2, S3, S4, noise, murmur, etc.) and finally, use our model in order to characterize real life segmented signals. Driven by this real life problem we have set new feature requirements for MetaProbLog's implementation.

[3] `https://dtai.cs.kuleuven.be/problog/`

[4] `https://www.circulodocoracao.com.br/sites/caravanadocoracao/en`

3 ProbLog Semantics

A ProbLog program T [4,6] consists of a set of facts annotated with probabilities $p_i :: pf_i$ – called *probabilistic facts* – together with a set of standard definite clauses $h :- b_1, \ldots, b_n.$ that can have positive and negative probabilistic literals in their body. A probabilistic fact pf_i is true with probability p_i. These facts correspond to random variables, which are assumed to be mutually independent. Together, they thus define a distribution over subsets of $L_T = \{pf_1, \ldots, pf_n\}$. The definite clauses add arbitrary *background knowledge* (BK) to those sets of *logical* facts. To keep a natural interpretation of a ProbLog program we assume that probabilistic facts cannot unify with other probabilistic facts or with the background knowledge rule heads.

Definition 1. *ProbLog Program: Formally, a ProbLog program is of the form* $T = \{pf_1, \ldots, pf_n\} \cup BK$.

Given the one-to-one mapping between ground definite clause programs and Herbrand interpretations, a ProbLog program defines a distribution over its Herbrand interpretations.

The distribution semantics are defined by generalising the least Herbrand models of the clauses by including subsets of the probabilistic facts. If fact pf_i is annotated with p_i, pf_i is included in a generalised least Herband model with probability p_i and left out with probability $1 - p_i$. The different facts are assumed to be probabilistically independent, however, negative probabilistic facts in clause bodies allow the user to enforce a choice between two clauses.

The MPE of T for a query q, is the most probable set $L_{MPE} \subseteq L_T$ of probabilistic facts (pf) or their negation contained at a randomly sampled subprogram d of T that entail q. The probability of the MPE is the product of the probabilities ($P(pf)$) of each probabilistic fact contained in L_{MPE} or $1.0 - P(pf)$ for contained probabilistic facts that are negated.

$$P(L_{MPE}) = \prod_{pf_i \in L_{true}} P(pf_i) \cdot \prod_{pf_j \in L_{false}} (1.0 - P(pf_j)) \tag{1}$$

where $L_{true} \cap L_{false} = L_{MPE}$. Finally, L_{MPE} is the set where:

$$argmax_{(L_{MPE} \in H_{interpretations})} P(L_{MPE}). \tag{2}$$

Because of the one-to-one mapping of explanations with Herbrand interpretations, the MPE is also the most likely Herbrand interpretation of the program T for the query q.

3.1 New Challenges

Originally, ProbLog did not supported general negation but only negated probabilistic facts. General negation was introduced by [8]. General negation introduces a new challenge for calculating the most probable witness of satisfiability.

The inference task for a negated subgoal is converted into finding the most probable witness of unsatisfiability.

A second challenge, originates in the introduction of Annotated Disjunctions (ADs) for ProbLog. ADs in ProbLog are compiled, by using a program transformation technique, to a set of probabilistic facts that through negation form a mutually exclusive structure. While this modification, does not affect most inference tasks, it makes the MPE humanly unreadable by returning the compiled probabilistic facts and not the ADs.

Finally, the addition of evidence in the new implementations of ProbLog impose a new challenge for computing the MPE. Evidence can be of an important use for our application as in some cases we might have a priori knowledge of some PCG signal characterizations. As shown and tackled in [14] ADs introduce a further complication for computing the MPE together with evidence.

3.2 MetaProbLog

MetaProbLog is an implementation of the ProbLog semantics within Yap Prolog [3]. In addition, MetaProbLog extends the semantics of ProbLog by defining a "ProbLog engine" which permits the definitions of probabilistic meta calls as presented in [10]. The "ProbLog engine" approach permits a more elegant handling of general negation for ProbLog, the use of any inference approaches as subgoals and the use of probabilistic meta calls. MetaProbLog inference, currently allows the computation of marginal probabilities with or without evidence and implements two inference methods **exact** and **program sampling**. Furthermore, it allows the computation of marginal probabilities for the answers of non-ground queries through the use of a special meta inference task called **ProbLog answers**. MetaProbLog has two unique features as a ProbLog implementation: probabilistic meta calls and datasets [9].

4 MetaProbLog's MPE for General ProbLog Programs and Future Support

While this work is still ongoing, we do have a preliminary implementation of ProbLog MPE inference that supports general negation. The original ProbLog implementation based its MPE inference on the fact that derivations are monotonic and that every added probabilistic fact will decrease the probability of the explanation. Because of that the original ProbLog MPE implementation was able to prune a big part of the search space.

Adding negated probabilistic facts (which the original ProbLog implementation supports) does not alter the MPE algorithm as the derivations remain monotonous decreasing. On the other hand general negation creates a significant complication. While the probability calculation remains monotonously decreasing and from the logic programming point of view the task remains the

same (find the most probable Herbrand interpretation), the search tree[5] traversed alters significantly. From a SAT point of view, general negation, converts the task in finding the most probable witness of unsastisfiability which is a hard problem [16].

For ProbLog programs without negation the search tree is composed by disjoint branches which are composed by conjoint literals (probabilistic facts). When general negation is applied on a subtree then by De Morgan's law the disjunctions are converted to conjunctions, the conjunctions to disjunctions and the literals become negated. In order to address this change on the search tree we are forced to collect all the negated goal's subtree and convert it by De Morgan's law. This however, delays the usage of the pruning mechanism which in some cases results to a computational overhead. Fortunately, this approach though does not increase the overall complexity of the inference algorithm. Furthermore, this mechanism is only activated for negated subgoals and not for the rest parts of the search tree.

While, currently we have already a first implementation of the described approach the next features are part of our work in progress.

4.1 Annotated Disjunction Representation

The existing representation for ADs creates a linear expansion of probabilistic facts for each AD value. This representation imposes two problems: first, the MPE returned to the user does not use the representation of the AD values instead it uses the linear expansion of probabilistic facts; second as shown in [14] the current representation computes wrongly the MPE in presence of evidence. We are working on a novel approach which uses the AD values directly for MPE and the linear representation only for other inference tasks solving both problems in a different way than the one presented in [14].

4.2 ProbLog Queries with Evidence

Furthermore, we need to address the conditional MPE task. In order to address this task we require a second search tree that provides the "evidence explanations (EEs)". Then we need to combine the candidate MPEs with the EEs in order to find the combination that maximizes the conditioned probability (for every probabilistic fact that is contained in EE, the probability of that fact does not contribute for the probability of the candidate explanation). Fortunately, the combinations can be pruned significantly.

4.3 Cycles and Tabling

Finally, in order for the algorithm and the implementation to fully support general ProbLog programs we need to be able to compute the MPE task in the presence of cycles. While this task is not needed for computing the MPE in

[5] For our task the search tree is the SLD tree used to prove the goal.

HMMs neither needed for our motivating application, it is necessary for the completeness of the resulting algorithm and implementation.

Cyclic programs in MetaProbLog are only allowed/handled with the conjuction of tabling (similarly as in Prolog). Handling this type of cycles has been already solved for the exact inference method in [8]. However, the combination of tabling and pruning is tricky imposing a different challenge. Possibly, MPE for cyclic programs will not be able to use pruning in the collection of explanations and only at a second stage, similarly to when handling general negation. ProbLog 2 MPE algorithm uses a similar strategy.

5 Conclusion

We presented the added challenges for calculating the MPE for general ProbLog programs in MetaProbLog. The added challenges compared with the original ProbLog are imposed by the new features of the new ProbLog implementations. Furthermore, we outlined how we intent to tackle these challenges. We require these features in order to apply them in our recent motivating application of computing the most probable characterization of PCG signals.

As we presented PCG signal characterization is an important motivating task that can aid in the diagnosis of heart diseases. We have already modeled the general behaviour of PCG signals in ProbLog as a HMM. We intent to grow our model to input real PCG signals and by using specific features of the signals to extract possible characterizations that later are been used as evidence in MPE inference in order to characterize the remaining signal features.

Acknowledgments: We want to thank the anonymous reviewers for their comments and help to improve our paper. Theofrastos Mantadelis is funded by the Portuguese Foundation for Science and Technology (FCT) within the project UID/EEA/50014/2013. Jorge Oliveira is funded by the Portuguese Foundation for Science and Technology (FCT) within the project HeartSafe, PTDC/EEI-PRO/2857/2012.

References

1. Bentley, P., Nordehn, G., Coimbra, M., Mannor, S.: The PASCAL classifying heart sounds challenge 2011 (CHSC2011) results., `http://www.peterjbentley.com/heartchallenge/`
2. Castro, A., Vinhoza, T., Mattos, S., Coimbra, M.: Heart sound segmentation of pediatric auscultations using wavelet analysis. In: Engineering in Medicine and Biology Society (EMBC). pp. 3909–3912 (July 2013)
3. Costa, V.S., Rocha, R., Damas, L.: The YAP prolog system. TPLP 12(1-2), 5–34 (2012)
4. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: IJCAI. pp. 2468–2473 (2007)

5. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theory and Practice of Logic Programming 15(03), 358–401 (May 2015), https://lirias.kuleuven.be/handle/123456789/392821
6. Kimmig, A., Demoen, B., Raedt, L.D., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. TPLP 11, 235–262 (2011)
7. Kimmig, A., De Raedt, L., Toivonen, H.: Probabilistic explanation based learning. In: European Conference on Machine Learning (ECML), vol. 4701, pp. 176–187 (2007)
8. Mantadelis, T., Janssens, G.: Dedicated tabling for a probabilistic setting. In: Hermenegildo, M.V., Schaub, T. (eds.) International Conference on Logic Programming (ICLP Technical Communications). LIPIcs, vol. 7, pp. 124–133. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
9. Mantadelis, T., Janssens, G.: MetaProbLog. ALP Newsletter (April 2015), http://www.cs.nmsu.edu/ALP/2015/04/metaproblog/
10. Mantadelis, T., Janssens, G.: Nesting probabilistic inference. CoRR abs/1112.3785 (2011), http://arxiv.org/abs/1112.3785
11. Moukadem, A., Dieterlen, A., Hueber, N., Brandt, C.: A robust heart sounds segmentation module based on s-transform. Biomedical Signal Processing and Control 8(3), 273 – 281 (2013)
12. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: International Conference on Logic Programming (ICLP). pp. 715–729. MIT Press (1995)
13. Sedighian, P., Subudhi, A., Scalzo, F., Asgari, S.: Pediatric heart sound segmentation using hidden markov model. In: Engineering in Medicine and Biology Society (EMBC). pp. 5490–5493 (Aug 2014)
14. Shterionov, D., Renkens, J., Vlasselaer, J., Kimmig, A., Meert, W., Janssens, G.: The most probable explanation for probabilistic logic programs with annotated disjunctions. In: International Conference on Inductive Logic Programming, Nancy, France, 14-16 September 2014 (2014), https://lirias.kuleuven.be/handle/123456789/474713
15. Viterbi, A.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. Information Theory 13(2), 260–269 (April 1967)
16. Wu, L., Zhou, H., Alava, M., Aurell, E., Orponen, P.: Witness of unsatisfiability for a random 3-satisfiability formula. CoRR abs/1303.2413 (2013), http://arxiv.org/abs/1303.2413
17. Ölmez, T., Dokur, Z.: Classification of heart sounds using an artificial neural network. Pattern Recognition Letters 24(1–3), 617 – 629 (2003)

Constraint-Based Inference in Probabilistic Logic Programs* (Extended Abstract)

Arun Nampally, C. R. Ramakrishnan

Department of Computer Science, Stony Brook University, Stony Brook, NY 11794
{anampally,cram}@cs.stonybrook.edu

1 Introduction

A wide variety of models that combine logical and statistical knowledge can be expressed succinctly in the Probabilistic Logic Programming (PLP) paradigm. Specifically, models in standard statistical formalisms such as probabilistic graphical models (PGMs) (e.g. Bayesian Networks), can be easily encoded as PLP programs. For instance, Fig. 1(a) shows a program in PRISM, a pioneering PLP language [16]. A widget, represented by a random variable X, is tested by two different processes b_1 and b_2. The outcomes of these tests are represented by random variables Y and Z, respectively. In PRISM, a special predicate of the form `msw(a,X)` associates random variable X with a random process `a`. Consider the problem of determining the distribution of X given that Y and Z are identical. Note that evidence is defined as a constraint over instantiations of the random variables, in contrast to a *specific* instantiation as in traditional PGMs. However, such evidence can be easily specified in PLP (see predicate `e/0`). The probability of a specific instantiation of X can also be computed based on a PLP query (e.g. `q(1)` using predicate `q/1`). This simple example illustrates how logical clauses can be used to specify evidence and queries in PLP that go beyond what is possible in traditional PGMs.

The Driving Problem. The expressiveness of PLP comes at a cost. Since PLP is an extension to traditional logic programming, inference in PLP is undecidable in general. Probabilistic inference for a large class of statistical models (e.g. Bayesian networks) is intractable. Even problems for which inference is tractable can be encoded in multiple ways in PLP, with different inference complexity. For instance, consider the PRISM program in Fig. 1(b). In that program, `genlist/2` defines a list of the outcomes of N identically distributed random variables ranging over `{a,b}`. Predicate `palindrome/1` tests, using a definite clause grammar definition, if a given list is a palindrome; and `count_as/2` tests if a given list contains k (not necessarily consecutive) "`a`"s. Using these predicates, consider the inference of the conditional probability of `query`(n, k) given `evidence`(n): *i.e.*, the probability that an n-element palindrome has k "`a`"s.

* This research was supported in part by NSF grants CCF-1018459 and IIS-1447549.

```
% Model: Y and Z depend on X.
w(X,Y,Z) :-
    msw(a, X),
    msw(b1(X), Y),
    msw(b2(X), Z).
% Evidence: Y and Z are same.
e :- w(_,S,S).
% Query: value of X.
q(X) :- w(X,_,_).
% Domains:
values(a, [1,2]).
values(b1(_), [1,2,3]).
values(b2(_), [2,3,4]).
% Distribution parameters:
set_sw(a, [0.4,0,6]).
set_sw(b1(1), [0.1,0.3,0.6]).
set_sw(b1(2), [0.2,0.4,0.4]).
set_sw(b2(1), [0.5,0.3,0.2]).
set_sw(b2(2), [0.6,0.1,0.3]).
```

(a) Bayesian Network PLP

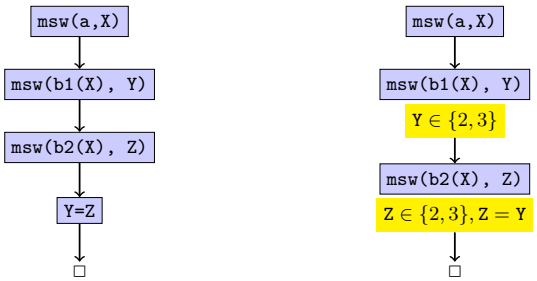
```
% generate a list of N random variables.
genlist(N, L) :- (N=0 -> L= []
                      ; msw(flip, N, X),
                        L = [X|L1], N1 is N-1,
                        genlist(N1, L1) ).
% Evidence: string is a palindrome.
evidence(N) :- genlist(N, L), palindrome(L).
% Query: string has K 'a's
query(N, K) :- genlist(N, L), count_as(L, K).
% Check if a given list is a palindrome
palindrome(L) :- phrase(palindrome, L).
palindrome --> [].
palindrome --> [_X].
palindrome --> [X], palindrome, [X].
% Query condition:
count_as([], 0).
count_as([X|Xs], K) :-
    K > 0, (X=a -> L is K-1; L=K),
    count_as(Xs, L).
% Domains:
values(flip, [a,b]).
% Distribution parameters:
set_sw(flip, [0.5, 0.5]).
```

(b) Palindrome PLP

Fig. 1: Examples of PLPs

The conditional probability is well-defined according to PRISM's distribution semantics [17]. However, *the PRISM itself will be unable to correctly compute the conditional query's probability*, since the conditional query, as encoded above, will violate the PRISM system's assumptions of independence among random variables used in an explanation. It should be noted that the above conditional probability may be efficiently inferred by transforming the "generate-and-test" program to one where the tests are folded into the generation phase. However, such transformations are dependent on the encoding of the query and evidence predicates, and are hard to generalize. Moreover, while the probability of goal `evidence(N)` can be computed in linear time (by exploiting sharing in explanation graphs), the size of the explanation graph for goal `query(N)` may be exponential in N when the subgoals in the explanations are placed in the order in which they are encountered.

Approximate inference based on rejection sampling performs poorly, rejecting a vast number of generated samples, since the likelihood of a string being a palindrome decreases exponentially in N. Alternatives such as Metropolis-Hastings-based Markov Chain Monte Carlo (MCMC) techniques [9, e.g.] do not behave much better due to the fact that the chains exhibit poor convergence (mixing), since most transitions lead to strings inconsistent with evidence. Gibbs-sampling-based MCMC [8] cannot be readily applied since the dependencies between random variables are hidden in the program and not explicit in the model.



(a) Before Constraint Propagation (b) After Constraint Propagation

Fig. 2: Symbolic Derivation for evidence "e" in BN Example

Our Approach. We identify two basic problems that contribute to the difficulty of inference in PLPs. First is that the random variable dependencies are not explicit in the program but may vary based on the program's control and data flow. The second is that evidence (and query) specifications may be complex rendering it difficult to predict whether a variable's valuation will be consistent with the evidence (or lead to query's success). We use *constraint propagation* both to uncover the hidden dependencies and to predict consistency with evidence. We explicitly construct *symbolic* derivations that abstract actual valuations of random variables and use a graphical structure to represent the derivations. We then provide inference algorithms, both approximate and exact, that compute the probability of a (possibly conditional) query based on this graphical structure.

Summary of Contributions. This paper describes a novel technique that addresses the problem of scalability of inference in PLPs.

1. The paper introduces a structure called an Ordered Symbolic Derivation Diagram to represent succinctly the set of possible derivations for a PLP query or evidence (Section 2).
2. The paper presents a likelihood-weighted sampling method based on OSDDs that can be used for approximate inference (Section 3).
3. The paper also presents an exact inference algorithm that operates directly on OSDDs. While this algorithm has relatively narrow applicability, it provides a powerful way to infer over large problem sizes without enumerating random variable valuations (Section 3).

We present experimental results which show the effectiveness of OSDD-based inference methods, as well as their cost (Section 4). Related work is discussed in detail in Section 5. The paper concludes with a discussion on the other uses of OSDDs for inference in PLPs.

2 Symbolic Derivations and Diagrams

In this paper, we use PRISM's syntax and distribution semantics, *but without the independence and mutual exclusion requirements on the explanations of a*

goal. Thus we consider PRISM programs with their *intended* model-theoretic semantics, rather than that computed by the PRISM system.

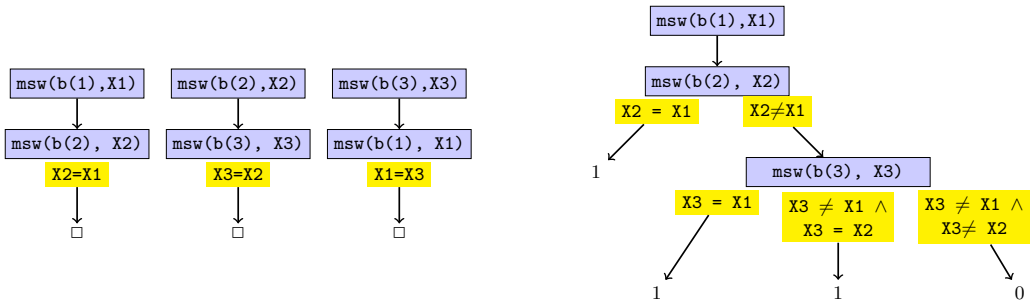
As stated in the Introduction, the dependencies between random variables implicit in the control and data flow of a PLP, and the impossibility of completely representing the set of all random variable valuations that are consistent with evidence contribute to the difficulty of inference in PLPs. To address these two problems, we devise an inference technique based on *constraint propagation*, to uncover hidden dependencies and predict evidence consistency. In the first step, we build derivations *symbolically*, instantiating random variables only when necessary. A symbolic derivation is a sequence of `msw` goals and constraints, as illustrated in Fig. 2(a) by the derivation for evidence "`e`" from example of Fig. 1(a).

In the second step, we propagate the constraints in a symbolic derivation, resulting in possible restrictions on the domains of variables. For instance, in the Bayesian Network (BN) example of Fig. 1(a), since the evidence demands $Y = Z$, the domains of b_1 and b_2 get restricted to `[2,3]`. Such constraint propagation is done using light-weight techniques such as node-consistency and arc-consistency algorithms. We add inferred domain restrictions (if any) to the derivation. We also place the constraints where their satisfaction can be effectively tested. Fig. 2(b) shows the symbolic derivation for "`e`" in the BN example after constraint propagation. The constraints denote a sufficient condition for any concrete instance of the symbolic derivation to represent a successful derivation. Symbolic derivations, parameterized by the consistency algorithms used in their construction, can be readily formalized; see [14]. Symbolic derivations can be subsequently used in a number of ways, two of which are described below.

Generalization. For many standard statistical models (e.g. PGMs) our technique will construct at most one symbolic derivation. In general, however, PLPs may have more than one symbolic derivation, as illustrated by the Birthday Collision example in Fig. 5. This example encodes the problem of determining the (unconditional) probability that two persons in a population of a given size share the same birthday. The query `same_birthday(3)`, which fixes a population of size 3, has 6 symbolic derivations, 3 of which are shown in Fig. 3(a). In such cases, we combine the set of symbolic derivations into a tree structure, called the Ordered Symbolic Derivation Diagram (OSDD), illustrated in Fig. 3(b). An OSDD is analogous to a Constraint Decision Diagram (CDD) [3]: each node defines a variable, and the outgoing edges are guarded by constraints on that variable. An OSDD is constructed based on a total order over variables, as in an Ordered Binary Decision Diagram [2]. Each path in an OSDD is a symbolic derivation. In fact, every symbolic derivation is a rudimentary OSDD (with 0-branches removed).

3 Inference Based on Symbolic Derivation Diagrams

We illustrate the process of generating likelihood-weighted samples [7,18] for goal "`e`" from its symbolic derivation. We start with likelihood weight of 1. When



(a) Selected Symbolic Derivations (b) Symbolic Derivation Diagram

Fig. 3: Symbolic Derivations for query "`same_birthday(3)`" in Birthday Collision Example

visiting `msw(a,X)`, we notice no domain restrictions on `X`, and hence bind `X` to a random sample generated from `a`'s distribution. Assume the sample we drew is `X=1`. We then visit `msw(b1(1), Y)` to sample `Y`. However, since `Y` has a domain restriction $Y \in \{2,3\}$, we generate a sample for `Y` such that $Y \in \{2,3\}$. This is done by picking from $\{2,3\}$ uniformly, and multiplying the likelihood weight with the probability of the picked value. Assume we pick `Y=2`; then the current likelihood is set to 0.3, the probability of 2 in `b1(1)`'s distribution. Finally, we visit `msw(b2(1),Z)`, whose two constraints restrict `Z` to $\{2\}$. Selecting `Z=2`, we multiply the likelihood weight of the current derivation with 0.5. Thus we generate a sample (`X=1`, `Y=2`, `Z=2`) consistent with evidence `e` with a likelihood weight of 0.15.

In summary, likelihood-weighted samples are drawn by (a) independently sampling random variables whose valuations are unconstrained; (b) uniformly sampling variables whose valuations have domain constraints; and (c) computing the probability of the sample as the product of probabilities of values picked in step (b). This procedure can be readily formalized; see [14].

Exact Inference. For certain class of programs and queries, symbolic derivations can be directly used for *exact* inference. Fig. 4 shows the symbolic derivation of evidence `evidence(6)` from the Palindrome example (Fig. 1(b)). Note that only the constraints in the symbolic derivation determine whether a concrete instance succeeds. Thus, if the distribution of `flip` is uniform, the three constraints are each satisfied independently, resulting in 0.125 as the probability. Such exact computation of probabilities is formalized in terms of *measurability*; a variable X with domain constraint η is said to be measurable if the size of X's domain (consistent with η) is independent of the valuation of other variables. When a symbolic derivation diagram consists only of uniformly distributed measurable variables, then the associated probability can be computed exactly. Such exact computation is readily formalized as well; see [14].

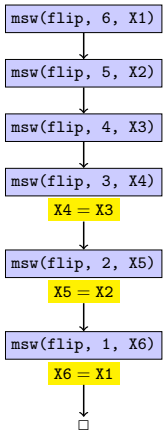


Fig. 4: Symbolic Derivation for evidence "`evidence(6)`" in Palindrome Example

```
% Two from a population of size N
% share a birthday.
same_birthday(N) :-
    person(N, P1),
    % P1's birthday is D
    msw(b(P1), D),
    person(N, P2),
    P1 \= P2,
    % and so is P2's.
    msw(b(P2), D).

person(N, P) :-
% bind P, backtracking through 1..N
    basics:for(P, 1, N).

% Distribution parameters:
set_sw(b(_), uniform(1,365)).
```

Fig. 5: Birthday Collision PLP

4 Experimental Evaluation

We present the results of experiments using a prototype implementation of a likelihood-weighted sampler based on symbolic derivations. The prototype uses XSB Prolog to build symbolic derivations, propagate constraints and construct OSDDs; and a few modules written in C for maintaining the sampler's state and dealing with random variable distributions. We used the following examples in the experiments.

- **Grid BN** is a Bayesian Network with Boolean random variables arranged in a 6×6 grid (with dependencies going left-to-right and top-to down). This simple structure was used to evaluate the effectiveness of our technique when the evidence probability is extremely low ($\tilde{}10^{-12}$).
- **Ising Model** is a well-known undirected graphical model. We used a 6×6 grid of Boolean random variables with factors on edges. The PRISM program independently generates values of terminal nodes of all edges, and ties them together by expressing equality constraints between shared variables of edges.
- **Palindrome**, which is shown in Fig. 1(b), with evidence limited to strings of length 20, and query checking for a string with 4 "`a`"s.
- **Birthday Collision**, shown in Fig. 5 (page 6), with population size of 6, i.e. query `same_birthday(6)`.

The first three examples involved conditional queries with low-likelihood evidence. The birthday collision example had an unconditional query. It should be noted that only the first example, **Grid BN**, can be evaluated in the PRISM system; the other examples have queries that violate PRISM's mutual exclusion and independence assumptions and hence cannot be directly evaluated in that

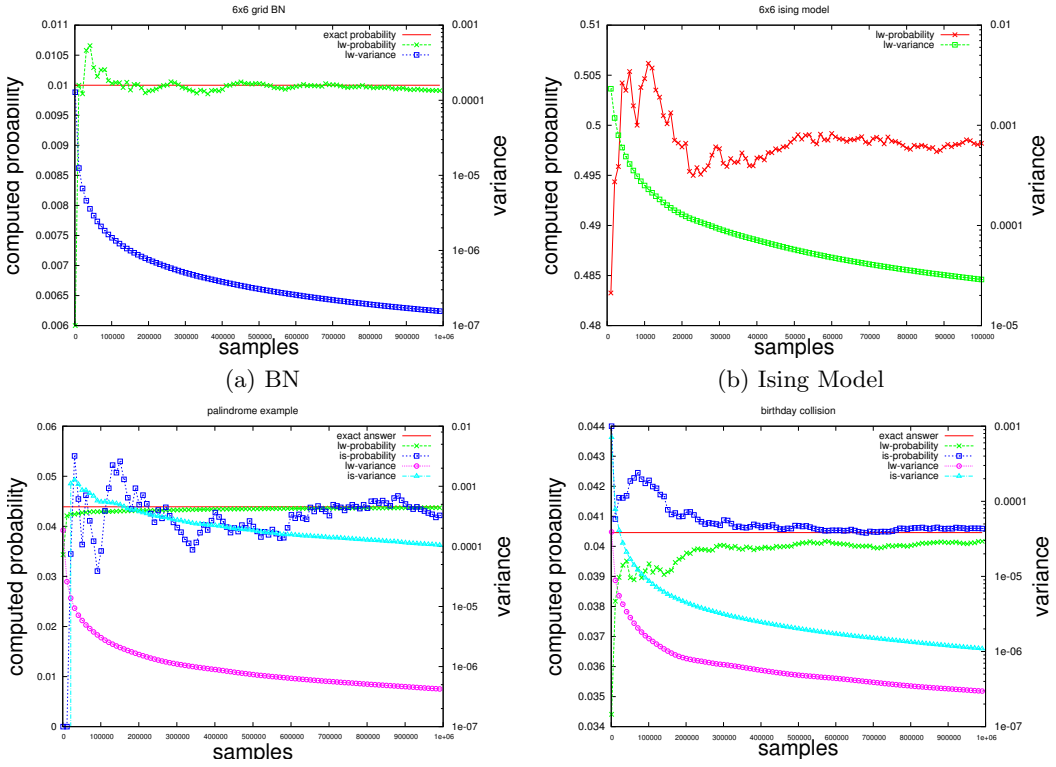


Fig. 6: Experimental Results

system. Our inference procedure, however, removes PRISM's assumptions and correctly evaluates the query probabilities for all the above examples.

The results of the experiments are shown in Fig. 6. Each subfigure plots the estimated probability and variance of the estimate(on log scale), for two samplers: the LW method described in this paper, and a simple independent sampler (with rejection sampling for conditional queries). Note that the LW sampler's results show significantly lower variance in all the examples. For the Grid BN and Ising Model, the evidence probability was low enough that a rejection sampler was unable to draw a consistent sample. The LW sampler, however, was able to converge to a reasonable estimate of low variance in about 500,000 samples. Both examples generated a single symbolic derivation. We directly sampled from this instead of materializing a OSDD structure. For the Grid BN, node consistency was sufficient to derive domain restrictions. For the Ising model, we found that standard LW sampling (picking a restricted value uniformly and assigning a likelihood weight) generated a number of samples with extremely low weights. Instead the probabilites of the set of allowed values were normalized to create a new proposal distribution. This resulted in generating samples with higher likelihood weights.

For the Palindrome example, we get a single symbolic derivation and the LW sampler quickly converges to the actual probability, while the independent sampler fails to converge even after a million samples. However, node- and arc-consistency can discover no further domain restrictions; forward checking at sampling time generated all the restrictions for LW sampler. The unusual pattern of variance for independent sampler in the initial iterations is due to it not being able to generate consistent samples and hence not having an estimate for the answer probability. The birthday collision example produces a number of symbolic derivations which were incorporated in an explicit OSDD. Domain restrictions are discovered only via forward checking, and that too for only one variable. The results show smaller difference between independent sampling and LW sampling for this example, compared to the others. One interesting observation from this example was that independent sampling using the OSDD structure was significantly faster (up to $2\times$) than using the program directly. This is because the program's non-deterministic evaluation has been replaced by a deterministic traversal through the OSDD.

Overheads. For all the examples, the time to construct the symbolic derivations, and propagate constraints was negligible (ranging from 4ms for Grid BN to 7ms for Birthday Collision, with XSB 3.5.0 on a 2.5GHz Intel Core 2 Duo machine). The overheads for sampling however were more pronounced. While an independent sampler picks values from the given distributions, the likelihood-weighting sampler needs to construct restricted domains to draw samples from. Consequently, our LW sampler takes up to $4\times$ per sample as an independent sampler.

Comparison with PITA and ProbLog. We evaluated the exact inference procedures of PITA and ProbLog on the same examples. We used a timeout of 15 minutes for both systems. The exact inference algorithm of ProbLog using *sentential decision diagrams* was able to handle Grid BN instances of size up to 9×9. However, ProbLog's inference does not scale beyond small problem sizes for the remaining three examples. In contrast, exact inference algorithm of PITA scaled much better. PITA could successfully compute the conditional probabilities for Grid BN (up to size 10×10), Ising model (up to 13×13) and Palindrome with $n = 18$. For the Ising model example, PITA's inference completes but with numerical errors due to the low probability of evidence. Finally, PITA's inference completed for the Birthday example with population size 2, but ran out of memory for larger population sizes.

5 Related Work

Probabilistic Constraint Logic Programming [11] extends PLP with constraint logic programming (CLP). It allows the specification of models with imprecise probabilities. Whereas a world in PLP denotes a specific assignment of values to random variables, a world in PCLP can define constraints on random variables,

rather than specific values. Lower and upper bounds are given on the probability of a query by summing the probabilities of worlds where query follows and worlds where query is possibly true respectively. While the way in which "proof constraints" of a PCLP query are obtained is similar to the way in which symbolic derivations are obtained (i.e., through constraint based evaluation), the inference techniques employed are completely different with PCLP employing satisfiability modulo theory (SMT) solvers.

cProbLog extends ProbLog with first-order constraints [6]. This gives the ability to express complex evidence in a succinct form. The semantics and inference are based on ProbLog. In contrast, our work makes the underlying constraints in a query explicit and uses the OSDDs to drive inference.

CLP($\mathcal{BN}$) [4] extends logic programming with constraints which encode conditional probability tables. A CLP($\mathcal{BN}$) program defines a joint distribution on the ground skolem terms. Operationally, queries are answered by constructing the relevant BN and performing BN inference.

There has been a significant interest in the area of lifted inference as exemplified by the work of [15,1,12]. The main idea of lifted inference is to treat indistinguishable *instances* random variables as one unit and perform inference at the population level. In contrast, exact inference using OSDDs treats indistinguishable *values* of random variables as one unit, thereby computing probabilities without grounding the random variables. Consequently, the method in this paper is orthogonal to traditional lifted inference and can be used when inversion and counting elimination are inapplicable (e.g. Birthday Collision example in Fig. 5).

The use of sampling methods for inference in PLPs has been widespread. The evidence has generally been handled by heuristics to reduce the number of rejected samples [5,13]. However we provide a systematic approach to deal with constraints imposed by evidence. When our constraint processing algorithm is powerful enough, the sampler can generate consistent samples without any rejections.

Adaptive sequential rejection sampling [10] is an algorithm that adapts its proposal distributions to avoid generating samples which are likely to be rejected. However, it requires a decomposition of the target distribution, which may not be available in PLPs. Further, in our work the distribution from which samples are generated is not adapted. It is an interesting direction of research to combine adaptivity with the proposed sampling algorithm.

6 Discussion

We presented a technique for inference in PLPs based on constructing a symbolic structure called OSDD using constraint propagation. The technique effectively performs inference without enumeration for a number of programs. The technique also uncovers the dependencies between random variables, which can then be exploited by more powerful inference techniques (e.g. Gibbs-sampling-based MCMC) that were inapplicable otherwise. However, for programs where sym-

bolic derivations match one-to-one with concrete derivations, the technique offers no benefit. An important topic of future work is to statically analyze a program to determine when (and when not to) use this technique. OSSDs are constructed by exploiting the presence of explicit random variables due to `msw`'s in PRISM. Application to other (equally expressive) PLP languages remains to be explored. Finally, OSDDs introduce a style of inference where indistinguishable valuations of random variables are treated together; combining this with lifted inference that groups indistinguishable random variables together will improve the scalability of inference in PLPs.

References

1. Rodrigo De Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 1319–1325, 2005.
2. Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.
3. Kenil CK Cheng and Roland HC Yap. Constrained decision diagrams. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 366. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.
4. Vítor Santos Costa, David Page, Maleeha Qazi, and James Cussens. CLP (BN): Constraint logic programming for probabilistic knowledge. In *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence*, pages 517–524. Morgan Kaufmann Publishers Inc., 2002.
5. James Cussens. Stochastic logic programs: Sampling, inference and applications. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 115–122. Morgan Kaufmann Publishers Inc., 2000.
6. Daan Fierens, Guy Van den Broeck, Maurice Bruynooghe, and Luc De Raedt. Constraints for probabilistic logic programming. In *Proceedings of the NIPS Probabilistic Programming Workshop*, pages 1–4, 2012.
7. Robert M. Fung and Kuo-Chu Chang. Weighing and integrating evidence for stochastic simulation in Bayesian Networks. In *Proceedings of the Fifth Annual Conference on Uncertainty in Artificial Intelligence*, UAI '89, pages 209–220, Amsterdam, The Netherlands, 1990. North-Holland Publishing Co.
8. Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (6):721–741, 1984.
9. W Keith Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.
10. Vikash K Mansinghka, Daniel M Roy, Eric Jonas, and Joshua B Tenenbaum. Exact and approximate sampling by systematic stochastic search. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics*, pages 400–407, 2009.
11. Steffen Michels, Arjen Hommersom, Peter JF Lucas, Marina Velikova, and Pieter Koopman. Inference for a new probabilistic constraint logic. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 2540–2546. AAAI Press, 2013.

12. Brian Milch, Luke S Zettlemoyer, Kristian Kersting, Michael Haimes, and Leslie Pack Kaelbling. Lifted probabilistic inference with counting formulas. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 1062–1068, 2008.
13. Bogdan Moldovan, Ingo Thon, Jesse Davis, and Luc De Raedt. MCMC estimation of conditional probabilities in probabilistic programming languages. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 436–448. Springer, 2013.
14. Arun Nampally and C. R. Ramakrishnan. Constraint-based inference in probabilistic logic programs. Technical report, Computer Science Department, Stony Brook University, `http://www.cs.stonybrook.edu/~cram/Papers/NR_LW15/lw15.pdf`, 2015.
15. David Poole. First-order probabilistic inference. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, volume 3, pages 985–991, 2003.
16. Taisuke Sato and Yoshitaka Kameya. PRISM: a language for symbolic-statistical modeling. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, volume 97, pages 1330–1339, 1997.
17. Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, pages 391–454, 2001.
18. Ross D. Shachter and Mark A. Peot. Simulation approaches to general probabilistic inference on belief networks. In *Proceedings of the Fifth Annual Conference on Uncertainty in Artificial Intelligence*, UAI '89, pages 221–234, Amsterdam, The Netherlands, 1990. North-Holland Publishing Co.

A Hybrid Approach to Inference in Probabilistic Non-Monotonic Logic Programming

Matthias Nickles and Alessandra Mileo

Insight Centre for Data Analytics
National University of Ireland, Galway
{matthias.nickles,alessandra.mileo}@deri.org

Abstract. We present a probabilistic inductive logic programming framework which integrates non-monotonic reasoning, probabilistic inference and parameter learning. In contrast to traditional approaches to probabilistic Answer Set Programming (ASP), our framework imposes only comparatively little restrictions on probabilistic logic programs - in particular, it allows for ASP as well as FOL syntax, and for precise as well as imprecise (interval valued) probabilities. User-configurable sampling and inference algorithms, which can be combined in a pipeline-like fashion, provide for general as well as specialized, more scalable approaches to uncertainty reasoning, allowing for adaptability with regard to different reasoning and learning tasks.

1 Introduction

With this paper, we present the probabilistic logic framework PrASP. PrASP is both a probabilistic logic programming language and a software system for probabilistic inference and inductive weight learning based on *Answer Set Programming* (ASP). Compared to previous works on this framework [11, 10], we introduce another inference algorithm and additional evaluation results.
Reasoning in the presence of uncertainty and relational structures such as social networks or Linked Data is an important aspect of knowledge discovery and representation for the Web, the Internet Of Things, and other heterogeneous and complex domains. Probabilistic logic programing, and the ability to learn probabilistic logic programs from data, can provide an attractive approach to uncertainty reasoning and statistical relational learning, since it combines the deduction power and declarative nature of logic programming with probabilistic inference abilities traditionally known from graphical models, such as Bayesian and Markov networks. We build upon existing approaches in the area of probabilistic (inductive) logic programming in order to provide a new ASP-based probabilistic logic programming language and inference tool which combines the benefits of non-monotonic reasoning using state-of-the-art ASP solvers with probabilistic inference and machine learning. The main enhancement provided by PrASP over (non-probabilistic) ASP as well as existing probabilistic approaches to ASP is the possibility to annotate any formula with point or interval (i.e., imprecise) probabilities (including formulas in full FOL syntax, albeit over finite

domains of discourse only), while providing a hybrid set of inference approaches: in addition to general inference algorithms, this includes specialized, more scalable inference algorithms for cases where certain optional assumptions hold (in particular mutual independence of probabilistic events). As we will show later, it can even make sense to combine different such algorithms (which can each on its own obtain valid results if its specific prerequisites are fulfilled).
The remainder of this paper is organized as follows: the next section presents related work. Section 3 describes the syntax and semantics of the formal framework. Section 4 describes approximate inference algorithms, and Section 5 provided initial evaluation results. Section 6 concludes.

2 Related Work

Approaches related to PrASP include [18, 8, 3–6, 14, 13, 15] which support probabilistic inference based on monotonic reasoning and [9, 1, 17, 2] which are based on non-monotonic logic programming. Like P-log [1], our approach computes probability distributions over answer sets (that is, possible worlds are identified with answer sets). However, P-log as well as [17] do not allow for annotating arbitrary formulas (including FOL formulas) with probabilities. [2] allows to associate probabilities with abducibles (only) and to learn both rules and probabilistic weights from given data (in form of literals). Again, PrASP does not impose such restrictions on probabilistic annotations or example data. On the other hand, PrASP cannot make use of abduction for learning. Various less closely related approaches to probabilistic reasoning exist (either not based on logic programming at all, or not in the realm of non-monotonic logic programming): Stochastic Logic Programs (SLP) [8] are an influential approach where sets of rules in form of range-restricted clauses can be labeled with probabilities. Parameter learning for SLPs is approached in [3] using the EM-algorithm. Approaches which combine concepts from Bayesian network theory with relational modeling and learning are, e.g., [4–6]. Probabilistic Relational Models (PRM) [4] can be seen as relational counterparts to Bayesian networks. In contrast to these, our approach does not directly relate to graphical models such as Bayesian or Markov Networks but works on arbitrary possible worlds which are generated by ASP solvers in form of stable models (answer sets). ProbLog [14] allows for probabilistic facts, annotated disjunctions and definite clauses, and approaches to probabilistic rule and parameter learning (from interpretations) also exist for ProbLog. ProbLog builds upon the Distribution Semantics approach introduced for PRISM [18], which is also used by other influential approaches, such as Independent Choice Logic (ICL) [13]. Another important approach outside the area of ASP are Markov Logic Networks (MLN) [15]. A Markov Logic Network consists of first-order formulas annotated with weights (which are, in contrast to PrASP, not in general probabilities). MLNs are used as templates for the construction of Markov networks. The (ground) Markov network generated from the MLN then determines a probability distribution over possible worlds, with inference performed using weighted SAT solving (which is related to but different

from ASP). MLNs are syntactically roughly similar to the logic programs in our framework (where weighted formulas can also be seen as soft or hard constraints for possible worlds).

3 Syntax and Semantics

In this section, we briefly describe the formal language and its semantics. Compared to [10], the syntax of PrASP programs has been extended (in particular by allowing interval and non-ground weights) and a variety of approximate inference algorithms have been added (see next section) to the default inference approach which is described below and which still underlies the formal semantics of PrASP programs.

PrASP is a Nilsson-style [12] probabilistic logic language. Let Φ be a set of function, predicate and object symbols and $\mathcal{L}(\Phi)$ a first-order language over Φ with the usual connectives (including both strong negation "-" and default negation "not") and first-order quantifiers. It can be assumed that this language covers both ASP and FOL syntax (ASP "specialties" such as choice constructs can be seen as syntactic sugar which we omit here in order to keep things simple). A PrASP program (background knowledge) is a non-empty finite set $\Lambda = \{[l_i; u_i]f_i\} \cup \{[l_i; u_i|c_i]f_i\} \cup \{indep(\{f_1^i, ..., f_n^i\})\}$ of annotated formulas (each concluded by a dot) and optional independence constraints (PrASP does not require an independence assumption but makes optionally use of declared or automatically discovered independence). $[l; u]f$ asserts that the imprecise probability of f is within interval $[l, u]$ (i.e., $l \leq Pr(f) \leq u$) whereas $[l; u|c]f$ states that the probability of f conditioned on formula c is within interval $[l, u]$ ($l \leq Pr(f|c) \leq u$).

Formulas can be non-ground (including existentially or universally quantified variables in FOL formulas). For the purpose of this paper, weights need to be ground (real numbers), however, the prototype implementation also allows for certain non-ground weights. An independence constraint $indep(\{f_1^i, ..., f_n^i\})$ specifies that the set of formulas $\{f_1^i, ..., f_n^i\}$ is mutually independent in the probabilistic sense (independence can also be discovered by PrASP by analyzing the background knowledge, but this is computationally more costly).

If the weight of a formula is omitted, $[1; 1]$ is assumed. Point probability weights $[p]$ are translated into weights of the form $[p; p]$ (analogously for conditional probabilities). Weighted formulas can intuitively be seen as constraints which specify which possible worlds (in the form of answer sets) are indeed possible, and with which probability. $w(f)$ denotes the weight of formula f. The f_i and c_i are formulas either in FOL syntax and supported by means of a transformation into ASP syntax described in [7]) or plain AnsProlog syntax, e.g., `[0.5] win :- coin(heads).` Informally, every FOL formula or program with FOL formulas results in a set of ASP formulas. The precise AnsProlog syntax depends on the external ASP grounder being employed by PrASP - in principle, any grounder could be used. The current prototype implementation has been tested with Gringo/Clingo 3 and 4 (`http://potassco.sourceforge.net`).

The semantics of PrASP is defined in terms of probability distributions over possible worlds which are identified with answer sets (models) - an assumption inspired by P-Log [1]. Let $M = (D, \Theta, \pi, \mu)$ be a probability structure where D is a finite discrete domain of objects, Θ is a non-empty set of possible worlds, π is a function which assigns to the symbols in Φ predicates, functions and objects over/from D, and $\mu = (\mu^l, \mu^u)$ is a discrete probability function over Θ, a PrASP program and a query formula, as defined further below.
Each possible world is a Herbrand interpretation over Φ. Since we will use answer sets (i.e., stable models of a (disjunctive) answer set program) as possible worlds, defining $\Gamma(a)$ to be the set of all answer sets of answer set program a will become handy.

We define a (non-probabilistic) satisfaction relation of possible worlds and unannotated programs as follows: let Λ^- be is an unannotated program and lp a transformation which transforms such a program (which might contain formulas in first-order logic syntax in addition to formulas in ASP syntax) into a disjunctive program. The details of this transformation are outside the scope of this paper and can be found in [7].

Then $(M, \theta) \models_\Theta \Lambda^-$ *iff* $\theta \in \Gamma(lp(\Lambda^-))$ and $\theta \in \Theta$. For a disjunctive program ψ, we define $(M, \theta) \models_\Theta \psi$ *iff* $\theta \in \Gamma(\psi)$ and $\theta \in \Theta$.

To do groundwork for the computation of a probability distribution over possible worlds Θ from a given PrASP program, we define a (non-probabilistic) satisfaction relation of possible worlds and unannotated formulas:
Let ϕ be a PrASP formula (without weight) and θ be a possible world. Furthermore, let $(M, \theta) \models_\Lambda \phi$ *iff* $(M, \theta) \models_\Theta \rho(\Lambda) \cup lp(\phi)$ *and* $\Theta = \Gamma(\rho(\Lambda))$ (we say formula ϕ is *true in possible world* θ). Sometimes we will just write $\theta \models_\Lambda \phi$ if M is given by the context. We abbreviate $(M, \theta) \models_\Lambda \phi$ as $\theta \models_\Lambda \phi$. At this, the *spanning program* $\rho(\Lambda)$ of PrASP program Λ is a non-probabilistic disjunctive program (without independence constraints) generated by removing all weights and transforming each formerly weighted formula f or $\neg f$ into a disjunction $f | \neg\, f$, where $\neg$ stands for default negation. Informally, the spanning program represents the uncertain but unweighted beliefs of the knowledge engineer or agent. With $\Gamma(a)$ as defined above, the set of possible worlds deemed possible according to existing belief $\rho(\Lambda)$ is denoted as $\Gamma(\rho(\Lambda))$.
We define the *minimizing* parameterized probability distribution $\mu^l(\Lambda, \Theta, q)$ over a set $\Theta = \{\theta_1, ..., \theta_m\} = \Gamma(\rho(\Lambda))$ of answer sets (possible worlds), a PrASP program $\Lambda = \{([p_i] f_i, i = 1..n)\} \cup \{([p_i | c_i] f_i^c)\} \cup \{indep(\{f_1^i, ..., f_k^i\})\}$ and a query formula q as $\{\theta_i \mapsto Pr(\theta_i) : \theta_i \in \Theta\}$ where $(Pr(\theta_1), ..., Pr(\theta_m))$ is any solution of the following system of inequalities (*constraints*) such that 1) $Pr^l(q) = \sum_{\theta_i \in \Theta : \theta_i \models_\Lambda q} Pr(\theta_i)$ is minimized and 2) the distribution has maximum entropy [19] among any other solutions which minimize the said sum. Analogously, μ^u denotes a maximum entropy probability distribution so that the $Pr(\theta_1), ..., Pr(\theta_m)$ maximize $Pr^u(q) = \sum_{\theta_i \in \Theta : \theta_i \models_\Lambda q} Pr(\theta_i)$.

$$l(f_1) \leq \sum_{\theta_i \in \Theta : \theta_i \models_\Lambda f_1} Pr(\theta_i) \leq u(f_1) \quad \cdots \quad l(f_n) \leq \sum_{\theta_i \in \Theta : \theta_i \models_\Lambda f_n} Pr(\theta_i) \leq u(f_n) \tag{1}$$

$$\sum_{\theta_i \in \Theta} \theta_i = 1 \tag{2}$$

$$\forall \theta_i \in \Theta : 0 \leq Pr(\theta_i) \leq 1 \tag{3}$$

At this, $l(f_i)$ and $u(f_i)$ denote the lower and upper endpoints of the probability interval (imprecise probability) of unconditional formula f_i (analogous for interval endpoints $l(f_i^c|c_i)$ and $u(f_i^c|c_i)$ of conditional probabilities).
In addition, any *indep*-declaration $indep(F^i)$ in the program induces for every subset $\{f_1^i, ..., f_r^i\} \subseteq F^i$, $r > 1$ constraints of the following form:
$\prod_{f_{k=1..r}^i} l(f_k^i) \leq \sum_{\theta_j \in \Theta : \theta_j \vDash_\Lambda \bigwedge f_{k=1..r}^i} Pr(\theta_j) \leq \prod_{f_{k=\{1..r\}}^i} u(f_k^i)$. In the case of point (i.e., precise) probabilities, these encode $Pr(\bigwedge_{k=1..r} f_k^i) = \prod_{k=1..r} Pr(f_k^i)$.
Furthermore, any conditional probability formula $[p_i|c_i]f_i^c)$ in the program induces constraints for ensuring $l(f_i^c|c_i) \leq Pr(f_i^c|c_i) \leq u(f_i^c|c_i)$
(with $p_i = [l(f_i^c|c_i); u(f_i^c|c_i)]$), namely
$\sum_{\theta_j \in \Theta} Pr(\theta_j)\nu(\theta_j, f_i^c \wedge c_i) + \sum_{\theta_j \in \Theta} -l(f_i^c|c_i)Pr(\theta_j)\nu(\theta_j, c_i) > 0$
$\sum_{\theta_j \in \Theta} Pr(\theta_j)\nu(\theta_j, f_i^c \wedge c_i) + \sum_{\theta_j \in \Theta} -u(f_i^c|c_i)Pr(\theta_j)\nu(\theta_j, c_i) < 0$
At this, we define $\nu(\theta, f) = \begin{cases} 1, & \text{if } \theta \vDash_\Lambda f \\ 0, & \text{otherwise} \end{cases}$
For small systems, PrASP can compute minimizing and maximizing probability distributions directly using the inequalities above with linear programming, and a maximum entropy solution amongst a number of candidate distributions (solutions of an underdetermined system) can be discovered using gradient descent. However, to make distribution finding tractable, we need to use different algorithms, as described in the next section. That is, the inequalities system above serves mainly as a means to define the semantics of PrASP formulas.
Finally, marginal inference results are obtained as follows: the result of a query of form `[?] q` is defined as the interval $[Pr^l(q), Pr^u(q)]$ and the result of conditional queries of form `[?|c] f` (which stands for $Pr(f|c)$, where c is some evidence) is computed using $Pr(f \wedge c)/Pr(c)$. An example PrASP program:

```
coin(1..10).
[0.4;0.6] coin_out(1,heads).
[[0.5]] coin_out(N,heads) :- coin(N), N != 1.
1{coin_out(N,heads), coin_out(N,tails)}1 :- coin(N).
n_win :- coin_out(N,tails), coin(N).
win :- not n_win.
[0.8|win] happy.
:- happy, not win.
```

The line starting with `[[0.5]]...` is syntactic sugar for a set of weighted rules where variable `N` is instantiated with all its possible values (i.e.,
`[0.5] coin_out(2,heads) :- coin(2), 2 != 1` and
`[0.5] coin_out(3,heads) :- coin(3), 3 != 1`). It would also be possible to use `[0.5]` as annotation of this rule, in which case the weight 0.5 would specify the probability of the entire non-ground formula instead.
`1{coin_out(N,heads), coin_out(N,tails)}1` (Gringo AnsProlog syntax) denotes that a coin comes up with either heads or tails but not both.

Our system accepts query formulas in format `[?] a`, which asks PrASP for the marginal probability of `a` and `[?|b] a` which computes the conditional probability $Pr(a|b)$. E.g., query `[?|coin_out(2,tails)] happy` results in [0;0].

4 Sampling and Inference Algorithms

PrASP (as a software system) contains a variety of exact and approximate inference algorithms which can be partially combined in a hybrid fashion. Using command line options, the user selects a *pipeline* of alternative pruning (simplification), sampling and inference steps (depending on the nature and complexity of the respective problem). E.g., the user might chose to sample possible worlds from a near-uniform distribution and to pass on the resulting models to a simulated annealing algorithm which computes a probability distribution over the sampled possible worlds. Finally, this distribution is used to compute the conditional or marginal probabilities of the query formulas. The inference algorithms available in the current prototype (version 0.7) of PrASP are:

Linear programming Direct solution for the linear inequalities system described before. Precise and very fast for very small systems, intractable otherwise.

Various answer set sampling algorithms for so-called *initial sampling* These can in some cases be used directly for inference, by computing a distribution which complies with the constraints (linear system) described before. An exemplary such algorithm is Algorithm 1. Alternatively, they can be followed by another inference algorithm (simulated annealing or iterative refinement, see below) which corrects the initial distribution computed by initial sampling.

Parallel simulated annealing This approach (Algorithm 2) performs simulated annealing for inference problems where no assumptions can be made about independence or other properties of the program (except consistency). It can be used either stand-alone or in a hybrid combination with an initial sampling stage (e.g., Algorithm 1).

Iterative refinement An adaptation of the inference algorithm described in [16] with guaranteed minimal Kullback−Leibler divergence to the uniform distribution (i.e., maximum entropy).

Direct counting Weights are transformed into unweighted formulas and queries are then solved by mere counting of models (see [10] for details).

Most of our algorithms rely heavily on near-uniform sampling, either using randomization provided by the respective external ASP solver (fast but typically rather low quality, i.e., weakly uniform) or using so-called XOR-constraints as described in [10] (which provides higher sampling quality at expense of speed).
From PrASP's inference algorithms, we describe one of the initial sampling algorithms (Algorithm 1) and parallel simulated annealing (Algorithm 2).
An interesting property of the first algorithm is its ability to provide a suitable distribution over possible worlds directly if all weighted formulas in the PrASP program are mutually independent (analogously to the independence assumption typically made by distribution semantics-based approaches). Algo. 2 can be

used stand-alone or subsequently to Algo. 1: in that case, the probability distribution computer by initial sampling (with replacement) is used as the initial distribution which is then refined by simulated annealing until all constraints (given probabilities) are fulfilled. The benefit of this pipelined approach to inference is that the user (knowledge engineer) doesn't need to know about event independence - if the uncertain formulas in the program are independent, initial sampling already provides a valid distribution and the subsequent simulated annealing stage almost immediately completes. Otherwise, simulated annealing "repairs" the insufficient distribution computed by the initial sampling stage.
Concretely, Algo. 1 samples answer sets and computes a probability distribution over these models which reflects the weights provided in the PrASP program, provided that all uncertain formulas in the program describe a mutually independent set of events. Other user-provided constraints (such as conditional probabilities in the PrASP program) are ignored here. Also, Algo. 1 does not guarantee that the solution has maximum entropy.

Algorithm 1 Sampling from models of spanning program (point probabilities only)

Require: max number of samples n, set of *uncertain* formulas $uf = \{[w(uf_i)]uf_i \text{ with } 0 < w(uf_i) < 1\}$, set of *certain* formulas $cf = \{cf_i : w(uf_i) = 1\}$ (i.e., with probability 1)

$i \leftarrow 1$
for $i \leq |uf|$ **do**
  $r^i \leftarrow$ random element of $Sym(\{1, ..., n\})$ (permutations of $\{1, ..., n\}$)
  $i \leftarrow i + 1$
end for
$m \leftarrow \emptyset$, $j \leftarrow 1$
parfor $j \in \{1, ..., n\}$ **do**
  $p \leftarrow \emptyset$, $k \leftarrow 1$
  for $k \leq |uf|$ **do**
    if $r^k_j \leq n \cdot w(uf_k)$ **then** $p \leftarrow p \cup \{uf_k\}$ **else** $p \leftarrow p \cup \{\neg uf_k\}$ **endif**
    $k \leftarrow k + 1$
  end for
  $s \leftarrow$ model sampled uniformly from models of program $cf \cup p$ ($\emptyset$ if UNSAT)
  $m \leftarrow m \uplus \{s\}$
end parfor

Ensure: Multiset m contains samples from all answer sets of spanning program such that

$\forall uf_i : w(uf_i) \approx \frac{|\{s \in m : s \models uf_i\}|}{|m|}$ *iff* set uf mutually independent.

Algorithm 2 presents the approach PrASP uses for approximate inference using a parallel form of simulated annealing (which does not require event independence). The initial list of samples *initSamples* are computed according to an initial sampling approach such as Algo. 1.

Algorithm 2 Inference by parallel simulated annealing Part 1.
We show only the basic variant for non-conditional formulas with point weights. The extension for conditional probabilities and interval probabilities (imprecise probabilities) is straightforward.

Require: *maxTime*, *maxEnergy*, *initTemp*, *initSamples* (from, e.g., Algo. 1). *initSamples* is a multiset which encodes a probability distribution via frequencies of models), *frozen*, *degreeOfParallelization*, α, F (set of weighted formulas), Λ (PrASP program)

$s \leftarrow initSamples$, $k \leftarrow 0$, $temp \leftarrow initTemp$
$e \leftarrow \text{ENERGY}(s)$
while $k \leq maxTime \wedge temp \geq frozen$ **do**
  parfor $i \leftarrow 1, degreeOfParallelization$ **do**
    $s''_i \leftarrow s'' \uplus \text{SAMPLESTEP}(samplingMethod)$
  end parfor
  $s' \leftarrow \text{argmin}_{s''}(\text{ENERGY}(s''_1), ..., \text{ENERGY}(s''_n))$
  $e' \leftarrow \text{ENERGY}(s')$
  if $e' < e \vee random_0^1 < e^{-(e'-e)/temp)}$ **then**
    $s \leftarrow s'$
    $e \leftarrow e'$
  end if
  $temp \leftarrow temp \cdot \alpha$
  $k \leftarrow k + 1$
end while

Ensure: Multiset $s = (pw, frq) = \mu_{approx}(\Lambda)$ approximates the probability distribution $\mu(\Lambda) = Pr(\Gamma(\rho(\Lambda)))$ over the set $pw = \{pw_i\} = \Gamma(\rho(\Lambda))$ of possible worlds by $\{Pr(pw_i) \approx \frac{frq(pw)}{|s|}\}$.

function ENERGY(s)
  parfor $f_i \in |F|$ **do**
    $freq_{f_i} \leftarrow \frac{|\{\{s' \in s : s' \models_\Lambda f_i\}\}|}{|s|}$
  end parfor
  return $\sqrt{\sum_{f_i \in F} (freq_{f_i} - weight_{f_i})^2}$
end function
▷ (Continued in Part 2 below)

In addition to the actual inference algorithms, it is often beneficial to let PrASP remove (prune) all parts of the spanning program which cannot influence the probabilities of the query formulas. The approach to this is straightforward (program dependency analysis) and therefore omitted here. We will show in the evaluation section how such simplification affects inference performance.

Algorithm 2 Inference by parallel simulated annealing Part 2.

22: **function** STEPSAMPLE(*samplingMethod*)
▷ The framework provides various configurable methods for the simulated annealing sampling step, of which we show here only one.
23: $F' \leftarrow \emptyset$
24: **for** $f_i \in |F|$ **do**
25: **if** $random_0^1 < weight_{f_i}$ **then**
26: $F' \leftarrow F' \cup \{f_i\}$
27: **else**
28: $F' \leftarrow F' \cup \{\neg f_i\}$
29: **end if**
30: **end for**
return $answerSets(F')$ (might be $\emptyset$)
31: **end function**

While for space-related reasons this paper covers deductive inference only, PrASP also supports induction (learning of weights of hypotheses from example data). Please refer to [10] for details.

5 Experiments

The main goal of PrASP is not to outperform existing approaches in terms of speed but to provide a flexible, scalable and highly configurable framework which puts as few restrictions as possible on what users can express in terms of (non-monotonic) certain and uncertain beliefs while being competitive with more specialized inference approaches if the respective conditions (like event independence) are met.
For our first experiment, we model a coin game (a slightly simplified variant of the example code shown before): a number of coins are tossed and the game is won if a certain subset of all coins comes up with "heads". The inference task is the approximation of the winning probability. In addition, another random subset of coins are magically dependent from each other and one of the coins is biased (probability of "heads" is 0.6). Despite its simplicity, this scenario shows how inference copes with independent as well as dependent uncertain facts, and how effective the pruning approach of the respective framework works (since winning depends only on a subset of coins). Also, inference complexity clearly scales with the number of coins. In PrASP syntax, such a partially randomly generated program looks, e.g., as follows (adaptation to MLN or ProbLog syntax is straightforward):

```
coin(1..8).
[0.6] coin_out(1,heads).
[[0.5]] coin_out(N,heads) :- coin(N), N != 1.
1{coin_out(N,heads), coin_out(N,tails)}1 :- coin(N).
win :- 2{coin_out(3,heads),coin_out(4,heads)}2.
coin_out(4,heads) :- coin_out(6,heads).
```

The inference algorithm used is initial sampling (Algo. 1) followed by simulated annealing (Algo. 2). Using Algo. 1, we computed 100 random models (number of samples n), which is sufficient to obtain a precision of +/-0.01 for the query probabilities. The winning subset of coins and the subset of mutually dependent coins (from which a rule of the form

```
coin_out(a,heads) :- coin_out(b,heads), coin_out(c,heads), ...
```

is generated) is each a random set with 25% of the size of the respective full set of coins. "PrASP 0.7.2 simp" in Fig. 1 stands for results obtained with pruning (i.e., parts of the program on which the query result cannot depend have been automatically removed). We also report the results obtained solely using (Algorithm 2) ("noinit" in Fig. 1), in order to see whether the initial sampling stage provides any benefits here. Simulated annealing parameters have been $maxEnergy = 0.15$, $initTemp = 5$, $frozen = 10^{-150}$, $\alpha = 0.85$.

We compared the performance (duration in dependency of the number of coins (x-axis), minimum number of 18 coins) of the current prototype of PrASP with that of Tuffy 0.3 (http://i.stanford.edu/hazy/hazy/tuffy/), a recent implementation of Markov Logic Networks which uses a database system in order to increase scalability, and ProbLog2 2.1 (https://dtai.cs.kuleuven.be/problog/) (despite random dependencies). Times are in milliseconds, obtained using an i7 4-cores processor with 3.4GHz over five trials.

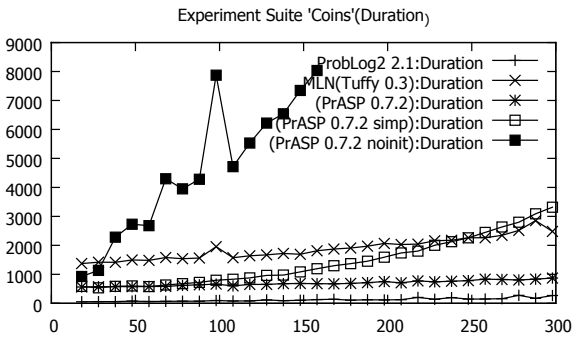


Fig. 1. Biased coins game

ProbLog2 and Tuffy scale very well here, with some additional time required by Tuffy probably due to the overhead introduced by external database operations. With PrASP, we observe that priming simulated annealing with an initial sampling step (which would give inaccurate results if used standalone) improves performance massively, whereas pruning appears to suffer from the additional overhead introduced by the required dependency analysis of the logic program. Our hypothesis regarding this behavior is that the initial distribution over possible worlds is, albeit not perfect, quite close to the accurate distribution so that the subsequent simulated annealing task takes off a lot faster compared to starting from scratch (i.e., from the uniform distribution).

The next experiment shows how PrASP copes with a more realistic benchmark task - a form of the well-known friends-and-smokers problem [15] - which can be tractably approached using Algorithm 1 alone since the independence assumption is met (which also makes it suitable for ProbLog). On the other hand, the rules are more complex. In this benchmark scenario, a randomly chosen number of persons are friends, a randomly chosen subset of all people

smoke, there is a certain probability for being stressed (`[[0.3]] stress(X)`), it is assumed that stress leads to smoking (`smokes(X) :- stress(X)`), and that some friends influence each other with a certain probability (`[[0.2]] influences(X,Y)`), in particular with regard to their smoking behavior `smokes(X) :- friend(X,Y), influences(Y,X), smokes(Y)`. With a certain probability, smoking leads to asthma (`[[0.4]] h(X). asthma(X) :- smokes(X), h(X)`). The query comprises of `[?] asthma(X)` for each person X.

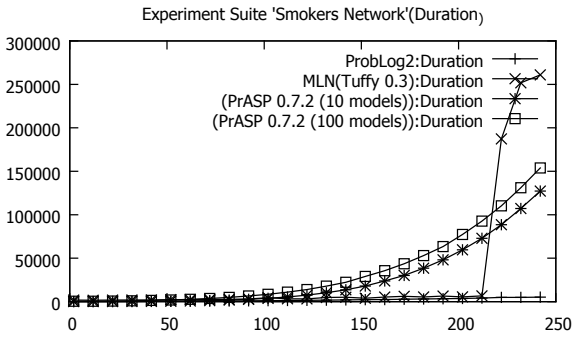


Fig. 2. Smokers social network

The results (Fig. 2) have been averaged over five trials. Again, ProbLog2 scores best in this scenario. PrASP, using Algorithm 1 (since all uncertain facts in this scenario are mutually independent), does quite well for most of episodes but looses on ProbLog2. Tuffy does very well below 212 persons, then performance massively breaks in for unknown reasons (possibly due to some internal cache overflow). For technical reasons, we couldn't get the smokers-scenario working with the publicly available current implementation of P-Log (we received segmentation faults which couldn't be resolved), but experiments with examples coming with this software seem to indicate that this approach also scales fine.

6 Conclusion

We have presented a new software framework for uncertainty reasoning and parameter estimation based on Answer Set Programming. In contrast to most other approaches to probabilistic logic programming, the philosophy of PrASP is to provide a very expressive formal language (ASP or full FOL syntax over finite domains for formulas annotated with precise as well as imprecise probabilities) on the one hand and a variety of inference algorithms which are able to take advantage of certain problem domains which facilitate "fast track" reasoning and learning (in particular inference in the presence of formula independence) on the other. We see the main benefit of our framework, besides its support for non-monotonic reasoning, thus in its semantically rich and configurable uncertainty reasoning approach which allows to combine various sampling and inference approaches in a pipeline-like fashion. Ongoing work focuses on additional experiments and the integration of further inference algorithms, and the direct integration of an ASP solver into PrASP, in order to avoid expensive calls of external reasoning tools. Another area of ongoing work is the support for so-called annotated disjunctions [20]. **Sponsored by SFI grant n. SFI/12/RC/2289.**

References

1. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. Theory Pract. Log. Program. 9(1), 57–144 (2009)
2. Corapi, D., Sykes, D., Inoue, K., Russo, A.: Probabilistic rule learning in nonmonotonic domains. In: Procs. 12th international conference on Computational logic in multi-agent systems. pp. 243–258. CLIMA'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2044543.2044565
3. Cussens, J.: Parameter estimation in stochastic logic programs. In: Mach. Learn. p. 2001 (2000)
4. Friedman, N., Getoor, L., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: In IJCAI. pp. 1300–1309. Springer-Verlag (1999)
5. Kersting, K., Raedt, L.D.: Bayesian logic programs. In: Proceedings of the 10th International Conference on Inductive Logic Programming (2000)
6. Laskey, K.B., Costa, P.C.: Of klingons and starships: Bayesian logic for the 23rd century. In: Procs. of the 21st Conf. on Uncertainty in Artificial Intelligence (2005)
7. Lee, J., Palla, R.: System f2lp - computing answer sets of first-order formulas. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR. Lecture Notes in Computer Science, vol. 5753, pp. 515–521. Springer (2009)
8. Muggleton, S.: Learning stochastic logic programs. Electron. Trans. Artif. Intell. 4(B), 141–153 (2000)
9. Ng, R.T., Subrahmanian, V.S.: Stable semantics for probabilistic deductive databases. Inf. Comput. 110(1), 42–83 (1994)
10. Nickles, M., Mileo, A.: Probabilistic inductive logic programming based on answer set programming. In: 15th Int'l Workshop on Non-Monotonic Reasoning (NMR'14) (2014)
11. Nickles, M., Mileo, A.: A system for probabilistic inductive answer set programming. In: 9th International Conference on Scalable Uncertainty Management (SUM'15) (2015, to appear)
12. Nilsson, N.J.: Probabilistic logic. Artificial Intelligence 28(1), 71–87 (1986)
13. Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artificial Intelligence 94, 7–56 (1997)
14. Raedt, L.D., Kimmig, A., Toivonen, H.: Problog: A probabilistic prolog and its application in link discovery. In: IJCAI. pp. 2462–2467 (2007)
15. Richardson, M., Domingos, P.: Markov logic networks. Mach. Learning 62(1-2), 107–136 (February 2006), http://dx.doi.org/10.1007/s10994-006-5833-1
16. Rodder, W., Meyer, C.: Coherent knowledge processing at maximum entropy by spirit. In: Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI'96), 1996 (1996)
17. Saad, E., Pontelli, E.: Hybrid probabilistic logic programming with non-monotoic negation. In: In Twenty First International Conference on Logic Programming. Springer Verlag (2005)
18. Sato, T., Kameya, Y.: Prism: a language for symbolic-statistical modeling. In: In Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI97. pp. 1330–1335 (1997)
19. Thimm, M., Kern-Isberner, G.: On probabilistic inference in relational conditional logics. Logic Journal of the IGPL 20(5), 872–908 (2012)
20. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: Demoen, B., Lifschitz, V. (eds.) Logic Programming, Lecture Notes in Computer Science, vol. 3132, pp. 431–445. Springer Berlin Heidelberg (2004)

The Distribution Semantics Is Well-Defined for All Normal Programs

Fabrizio Riguzzi

Dipartimento di Matematica e Informatica, Università di Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
fabrizio.riguzzi@unife.it

Abstract. The distribution semantics is an approach for integrating logic programming and probability theory that underlies many languages and has been successfully applied in many domains. When the program has function symbols, the semantics was defined for special cases: either the program has to be definite or the queries must have a finite number of finite explanations. In this paper we show that it is possible to define the semantics for all programs.

Keywords: Distribution Semantics, Function Symbols, ProbLog, Probabilistic Logic Programming

1 Introduction

The distribution semantics [1, 2] was successfully applied in many domains and underlies many languages that combine logic programming with probability theory such as Probabilistic Horn Abduction, Independent Choice Logic, PRISM, Logic Programs with Annotated Disjunctions and ProbLog.

The definition of the distribution semantics can be given quite simply in the case of no function symbols in the program: a probabilistic logic program under the distribution semantics defines a probability distribution over normal logic programs called *worlds* and the probability of a ground query can be obtained by marginalizing the joint distribution of the worlds and the query. In the case the program has function symbols, however, this simple definition does not work as the probability of individual worlds is zero.

A definition of the distribution semantics for programs with function symbols was proposed in [1, 3] but restricted to definite programs. The case of normal programs was taken into account in [4] where the semantics required that the programs are acyclic. A looser condition was proposed in [5] but still required each goal to have a finite set of finite explanations.

In this paper we show that the distribution semantics can be defined for all programs, thus also for programs that have goals with an infinite number of possibly infinite explanations. We do so by adapting the definition of the well-founded semantics in terms of iterated fixpoints of [6] to the case of ProbLog, similarly to the way in which the T_P operator has been adapted in [7] to the

case of stratified ProbLog programs using parameterized interpretations. In the case of infinite number of infinite explanations, we show that the probability of queries is defined in the limit and the limit always exists.

We consider the case of ProbLog but the results are equally applicable to all other languages under the distribution semantics, as there are linear transformations from one language to another that preserve the semantics.

The paper is organized as follows. Section 2 presents preliminary material on fixpoints and the well-founded semantics. Section 3 introduces the distribution semantics for programs without function symbols. Section 4 discusses the definition of the distribution semantics with function symbols in the case of finite set of finite explanations. Section 5 represents the main contribution of this paper and discusses the case of infinite set of infinite explanations. Finally, Section 6 concludes the paper. The proofs of the main results are reported in the Appendix.

2 Preliminaries

A relation on a set S is a *partial order* if it is reflexive, antisymmetric and transitive. In the following, let S be a set with a partial order $\leq$. $a \in S$ is an *upper bound* of a subset X of S if $x \leq a$ for all $x \in X$. Similarly, $b \in S$ is a *lower bound* of X if $b \leq x$ for all $x \in X$.

$a \in S$ is the *least upper bound* of a subset X of X if a is an upper bound of X and, for all upper bounds a' of X, we have $a \leq a'$. Similarly, $b \in S$ is the *greatest lower bound* of a subset X of S if b is a lower bound of X and, for all lower bounds b' of X, we have $b' \leq b$. The least upper bound of X is unique, if it exists, and is denoted by $lub(X)$. Similarly, the greatest lower bound of X is unique, if it exists, and is denoted by $glb(X)$.

A partially ordered set L is a *complete lattice* if $lub(X)$ and $glb(X)$ exist for every subset X of L. We let $\top$ denote the top element $lub(L)$ and $\bot$ denote the bottom element $glb(L)$ of the complete lattice L.

Let L be a complete lattice and $T : L \to L$ be a mapping. We say T is *monotonic* if $T(x) \leq T(y)$, whenever $x \leq y$. We say $a \in L$ is the *least fixpoint* of T if a is a fixpoint (that is, $T(a) = a$) and for all fixpoints b of T we have $a \leq b$. Similarly, we define *greatest fixpoint*.

Let L be a complete lattice and $T : L \to L$ be monotonic. Then we define $T \uparrow 0 = \bot$; $T \uparrow \alpha = T(T \uparrow (\alpha - 1))$, if α is a successor ordinal; $T \uparrow \alpha = lub(\{T \uparrow \beta | \beta < \alpha\})$, if α is a limit ordinal; $T \downarrow 0 = \top$; $T \downarrow \alpha = T(T \downarrow (\alpha - 1))$, if α is a successor ordinal; $T \downarrow \alpha = glb(\{T \downarrow \beta | \beta < \alpha\})$, if α is a limit ordinal.

Proposition 1. *Let L be a complete lattice and $T : L \to L$ be monotonic. Then T has a lest fixpoint, $lfp(T)$ and a greatest fixpoint $gfp(T)$.*

A normal program P is a set of normal rules. A normal rule has the form

$$r = h \leftarrow b_1, \ldots, b_n, not\, c_1, \ldots, not\, c_m \tag{1}$$

where $h, b_1, \ldots, b_n, c_1, \ldots, c_m$ are atoms.

The set of ground atoms that can be built with the symbols of a program P is called the Herbrand base and is denoted as $\mathcal{B}_P$.

A *two-valued interpretation* I is a subset of $\mathcal{B}_P$. I is the set of true atoms. The set $Int2$ of two-valued interpretations for a program P forms a complete lattice where the partial order $\leq$ is given by the subset relation $\subseteq$. The least upper bound and greatest lower bound are defined as $lub(X) = \bigcup_{I \in X} I$ and $glb(X) = \bigcap_{I \in X} I$. The bottom and top element are respectively $\emptyset$ and $\mathcal{B}_P$.

A *three-valued interpretation* $\mathcal{I}$ is a pair $\langle I_T; I_F \rangle$ where I_T and I_F are subsets of $\mathcal{B}_P$ and represent respectively the set of true and false atoms. The union of two three-valued interpretations $\langle I_T, I_F \rangle$ and $\langle J_T, J_F \rangle$ is defined as $\langle I_T, I_F \rangle \cup \langle J_T, J_F \rangle = \langle I_T \cup J_T, I_F \cup J_F \rangle$. The intersection of two three-valued interpretations $\langle I_T, I_F \rangle$ and $\langle J_T, J_F \rangle$ is defined as $\langle I_T, I_F \rangle \cap \langle J_T, J_F \rangle = \langle I_T \cap J_T, I_F \cap J_F \rangle$.

The set $Int3$ of three-valued interpretations for a program P forms a complete lattice where the partial order $\leq$ is defined as $\langle I_T, I_F \rangle \leq \langle J_T, J_F \rangle$ if $I_T \subseteq J_T$ and $I_F \subseteq J_F$. The least upper bound and greatest lower bound are defined as $lub(X) = \bigcup_{I \in X} I$ and $glb(X) = \bigcap_{I \in X} I$. The bottom and top element are respectively $\langle \emptyset, \emptyset \rangle$ and $\langle \mathcal{B}_P, \mathcal{B}_P \rangle$.

The well-founded semantics (WFS) assigns a three-valued model to a program, i.e., it identifies a three-valued interpretation as the meaning of the program. The WFS was given in [8] in terms of the least fixpoint of an operator that is composed by two sub-operators, one computing consequences and the other computing unfounded sets. We give here the alternative definition of the WFS of [6] that is based on a different iterated fixpoint.

Definition 1. *For a normal program P, sets Tr and Fa of ground atoms, and a 3-valued interpretation $\mathcal{I}$ we define*

$OpTrue^P_{\mathcal{I}}(Tr) = \{a | a$ *is not true in* $\mathcal{I}$*; and there is a clause* $b \leftarrow l_1, ..., l_n$ *in* P*, a grounding substitution* θ *such that* $a = b\theta$ *and for every* $1 \leq i \leq n$ *either* $l_i\theta$ *is true in* $\mathcal{I}$*, or* $l_i\theta \in Tr\}$*;*

$OpFalse^P_{\mathcal{I}}(Fa) = \{a | a$ *is not false in* $\mathcal{I}$*; and for every clause* $b \leftarrow l_1, ..., l_n$ *in* P *and grounding substitution* θ *such that* $a = b\theta$ *there is some* i $(1 \leq i \leq n)$ *such that* $l_i\theta$ *is false in* $\mathcal{I}$ *or* $l_i\theta \in Fa\}$*.*

In words, the operator $OpTrue^P_{\mathcal{I}}$ extends the interpretation $\mathcal{I}$ to add the new atomic facts that can be derived from P knowing $\mathcal{I}$, while $OpFalse^P_{\mathcal{I}}$ adds the new negations of atomic facts that can be shown false in P by knowing $\mathcal{I}$. $OpTrue^P_{\mathcal{I}}$ and $OpFalse^P_{\mathcal{I}}$ are both monotonic [6], so they both have a least and greatest fixpoints. An iterated fixpoint operator builds up *dynamic strata* by constructing successive three-valued interpretations as follows.

Definition 2 (Iterated Fixed Point). *For a normal program P, let $IFP^P : Int3 \rightarrow Int3$ be defined as $IFP^P(\mathcal{I}) = \mathcal{I} \cup \langle lfp(OpTrue^P_{\mathcal{I}}), gfp(OpFalse^P_{\mathcal{I}}) \rangle$.*

IFP^P is monotonic [6] and thus as a least fixed point $lfp(IFP^P)$. Moreover, the well-founded model $WFM(P)$ of P is in fact $lfp(IFP^P)$. Let δ be the smallest ordinal such that $WFM(P) = IFP^P \uparrow \delta$. We refer to δ as the *depth* of P. The *stratum* of atom a is the least ordinal β such that $a \in IFP^P \uparrow \beta$ (where a may

be either in the true or false component of $IFP^P \uparrow \beta$). Undefined atoms of the well-founded model do not belong to any stratum – i.e. they are not added to $IFP^P \uparrow \delta$ for any ordinal δ.

3 The Distribution Semantics for Programs without Function Symbols

We present the distribution semantics for the case of ProbLog [9] as it is the language with the simplest syntax. A ProbLog program $\mathcal{P}$ is composed by a set of normal rules $\mathcal{R}$ and a set $\mathcal{F}$ of probabilistic facts. Each *probabilistic fact* is of the form $p_i :: a_i$ where $p_i \in [0, 1]$ and a_i is an atom, meaning that each ground instantiation $a_i\theta$ of a_i is true with probability p_i and false with probability $1 - p_i$. Each world is obtained by selecting or rejecting each grounding of all probabilistic facts.

An *atomic choice* is the selection or not of grounding $F\theta$ of a probabilistic fact F. It is represented with the triple (F, θ, i) where $i \in \{0, 1\}$. A set κ of atomic choices is *consistent* if it does not contain two atomic choices (F, θ, i) and (F, θ, j) with $i \neq j$ (only one alternative is selected for a ground probabilistic fact). The function $consistent(\kappa)$ returns true if κ is consistent. A *composite choice* κ is a consistent set of atomic choices. The probability of composite choice κ is $P(\kappa) = \prod_{(F_i,\theta,1)\in\kappa} p_i \prod_{(F_i,\theta,0)\in\kappa} 1 - p_i$ where p_i is the probability of the i-th probabilistic fact F_i. A *selection* σ is a total composite choice, i.e., it contains one atomic choice for every grounding of each probabilistic fact. A *world* w_σ is a logic program that is identified by a selection σ. The world w_σ is formed by including the atom corresponding to each atomic choice $(F, \theta, 1)$ of σ.

The probability of a world w_σ is $P(w_\sigma) = P(\sigma)$. Since in this section we are assuming programs without function symbols, the set of groundings of each probabilistic fact is finite, and so is the set of worlds $W_\mathcal{P}$. Accordingly, for a ProbLog program $\mathcal{P}$, $W_\mathcal{P} = \{w_1, \ldots, w_m\}$. Moreover, $P(w)$ is a distribution over worlds: $\sum_{w\in W_\mathcal{P}} P(w) = 1$. We call *sound* a program for which every world has a two-valued well-founded model. We consider only sound programs, as the uncertainty should be handled by the choices rather than by the semantics of negation.

Let q be a query in the form of a ground atom. We define the conditional probability of q given a world w as: $P(q|w) = 1$ if q is true in w and 0 otherwise. Since the program is sound, q can be only true or false in a world. The probability of q can thus be computed by summing out the worlds from the joint distribution of the query and the worlds: $P(q) = \sum_w P(q, w) = \sum_w P(q|w)P(w) = \sum_{w\models q} P(w)$.

4 The Distribution Semantics for Programs with Function Symbols

When a program contains functions symbols there is the possibility that its grounding may be infinite. If so, the number of atomic choices in a selection that

defines a world is countably infinite and there is an uncountably infinite number of worlds. In this case, the probability of each individual world is zero since it is the product of infinite numbers all smaller than one. So the semantics of Section 3 is not well-defined.

Example 1. Consider the program

$$\begin{array}{lll} p(0) \leftarrow u(0). & t \leftarrow \neg s. & F_1 = a :: u(X). \\ p(s(X)) \leftarrow p(X), u(X). & s \leftarrow r, \neg q. & F_2 = b :: r. \\ & q \leftarrow u(X). & \end{array}$$

The set of worlds is infinite and uncountable. In fact, each world can be put in a one to one relation with a selection and a selection can be represented as a countable sequence of atomic choices of which the first involves fact F_2, the second $F_1/\{X/0\}$, the third $F_1/\{X/s(0)\}$ and so on. The set of selections can be shown uncountable by Cantor's diagonal argument. Suppose the set of selections is countable. Then the selections could be listed in order, suppose from top to bottom. Suppose the atomic choices of each selection are listed from left to right. We can pick a composite choice that differs from the first selection in the first atomic choice (if $(F_2, \emptyset, k)$ is the first atomic choice of the first selection, pick $(F_2, \emptyset, 1-k)$), from the second selection in the second atomic choice (similar to the case of the first atomic choice) and so on. In this way we have obtained a selection that is not present in the list because it differs from each selection in the list for at least an atomic choice. So it is not possible to list the selections in order.

We now present the definition of the distribution semantics for programs with function symbols following [4]. The semantics for a probabilistic logic program $\mathcal{P}$ with function symbols is given by defining a *probability measure* μ over the set of worlds $W_\mathcal{P}$. Informally, μ assigns a probability to a set of *subsets* of $W_\mathcal{P}$, rather than to every element of (the infinite set) $W_\mathcal{P}$. The approach dates back to [10] who defined a probability measure μ as a real-valued function whose domain is a *σ-algebra* Ω on a set $\mathcal{W}$ called the *sample space*. Together $\langle \mathcal{W}, \Omega, \mu \rangle$ is called a *probability space*.

Definition 3. *[11, Section 3.1] The set Ω of subsets of $\mathcal{W}$ is a* σ-algebra *on the set $\mathcal{W}$ iff (σ-1) $\mathcal{W} \in \Omega$; (σ-2) Ω is closed under complementation, i.e., $\omega \in \Omega \rightarrow (\mathcal{W} \setminus \omega) \in \Omega$; and ($\sigma$-3) Ω is closed under countable union, i.e., if $\omega_i \in \Omega$ for $i = 1, 2, \ldots$ then $\bigcup_i \omega_i \in \Omega$.*

The elements of Ω are called *measurable sets*. Importantly, for defining the distribution semantics for programs with function symbols, not every subset of $\mathcal{W}$ need be present in Ω.

Definition 4. *[10] Given a sample space $\mathcal{W}$ and a σ-algebra Ω of subsets of $\mathcal{W}$, a* probability measure *is a function $\mu : \Omega \rightarrow \mathbb{R}$ that satisfies the following axioms: (μ-1) $\mu(\omega) \geq 0$ for all $\omega \in \Omega$; (μ-2) $\mu(\mathcal{W}) = 1$; (μ-3) μ is countably additive, i.e., if $O = \{\omega_1, \omega_2, \ldots\} \subseteq \Omega$ is a countable collection of pairwise disjoint sets, then $\mu(\bigcup_{\omega \in O}) = \sum_i \mu(\omega_i)$.*

We first consider the finite additivity version of probability spaces. In this stronger version, the σ-algebra is replaced by an algebra.

Definition 5. *[11, Section 3.1] The set Ω of subsets of $\mathcal{W}$ is an* algebra *on the set $\mathcal{W}$ iff it respects conditions (σ-1), (σ-2) and condition (a-3): Ω is closed under finite union, i.e., $\omega_1 \in \Omega, \omega_2 \in \Omega \rightarrow (\omega_1 \cup \omega_2) \in \Omega$*

The probability measure is replaced by a finitely additive probability measure.

Definition 6. *Given a sample space $\mathcal{W}$ and an algebra Ω of subsets of $\mathcal{W}$, a* finitely additive probability measure *is a function $\mu : \Omega \rightarrow \mathbb{R}$ that satisfies axioms (μ-1) and (μ-2) of Definition 4 and axiom (m-3): μ is finitely additive, i.e., $\omega_1 \cap \omega_2 = \emptyset \rightarrow \mu(\omega_1 \cup \omega_2) = \mu(\omega_1) + \mu(\omega_2)$ for all $\omega_1, \omega_2 \in \Omega$.*

Towards defining a suitable algebra given a probabilistic logic program $\mathcal{P}$, we define the *set of worlds ω_κ compatible with* a composite choice κ as $\omega_\kappa = \{w_\sigma \in W_\mathcal{P} | \kappa \subseteq \sigma\}$. Thus a composite choice identifies a set of worlds. For programs without function symbols $P(\kappa) = \sum_{w \in \omega_\kappa} P(w)$.

Given a *set* of composite choices K, the *set of worlds ω_K compatible with K* is $\omega_\mathcal{K} = \bigcup_{\kappa \in K} \omega_\kappa$. Two composite choices κ_1 and κ_2 are *incompatible* if their union is not consistent. A set K of composite choices is *pairwise incompatible* if for all $\kappa_1 \in K, \kappa_2 \in K$, $\kappa_1 \neq \kappa_2$ implies that κ_1 and κ_2 are incompatible.

Regardless of whether a probabilistic logic program has a finite number of worlds or not, obtaining pairwise incompatible sets of composite choices is an important problem. This is because the *probability of a pairwise incompatible set K of composite choices* is defined as $P(K) = \sum_{\kappa \in K} P(\kappa)$ which is easily computed. Two sets K_1 and K_2 of finite composite choices are *equivalent* if they correspond to the same set of worlds: $\omega_{K_1} = \omega_{K_2}$.

One way to assign probabilities to a set K of composite choices is to construct an equivalent set that is pairwise incompatible; such a set can be constructed through the technique of *splitting*. More specifically, if $F\theta$ is an instantiated fact and κ is a composite choice that does not contain an atomic choice (F, θ, k) for any k, the *split* of κ on $F\theta$ is the set of composite choices $S_{\kappa,F\theta} = \{\kappa \cup \{(F, \theta, 0)\}, \kappa \cup \{(F, \theta, 1)\}\}$. It is easy to see that κ and $S_{\kappa,F\theta}$ identify the same set of possible worlds, i.e., that $\omega_\kappa = \omega_{S_{\kappa,F\theta}}$, and that $S_{\kappa,F\theta}$ is pairwise incompatible. The technique of splitting composite choices on formulas is used for the following result [12].

Theorem 1 *(Existence of a pairwise incompatible set of composite choices [12]) Given a finite set K of composite choices, there exists a finite set K' of pairwise incompatible composite choices such that K and K' are equivalent.*

Proof: Given a finite set of composite choices K, there are two possibilities to form a new set K' of composite choices so that K and K' are equivalent:

1. **removing dominated elements**: if $\kappa_1, \kappa_2 \in K$ and $\kappa_1 \subset \kappa_2$, let $K' = K \setminus \{\kappa_2\}$.
2. **splitting elements**: if $\kappa_1, \kappa_2 \in K$ are compatible (and neither is a superset of the other), there is a $(F, \theta, k) \in \kappa_1 \setminus \kappa_2$. We replace κ_2 by the split of κ_2 on $F\theta$. Let $K' = K \setminus \{\kappa_2\} \cup S_{\kappa_2,F\theta}$.

In both cases $\omega_K = \omega_{K'}$. If we repeat this two operations until neither is applicable we obtain a splitting algorithm that terminates because K is a finite set of composite choices. The resulting set K' is pairwise incompatible and is equivalent to the original set. ♢

Theorem 2 *(Equivalence of the probability of two equivalent pairwise incompatible finite set of finite composite choices [13]) If K_1 and K_2 are both pairwise incompatible finite sets of finite composite choices such that they are equivalent then $P(K_1) = P(K_2)$.*

For a probabilistic logic program $\mathcal{P}$, we can thus define a unique probability measure $\mu : \Omega_\mathcal{P} \to [0, 1]$ where $\Omega_\mathcal{P}$ is defined as the set of sets of worlds identified by finite sets of finite composite choices: $\Omega_\mathcal{P} = \{\omega_K | K$ is a finite set of finite composite choices $\}$. $\Omega_\mathcal{P}$ is an algebra over $W_\mathcal{P}$ since $W_\mathcal{P} = \omega_K$ with $K = \{\emptyset\}$. Moreover, the complement ω_K^c of ω_K where K is a finite set of finite composite choice is $\omega_{\overline{K}}$ where $\overline{K}$ is a finite set of finite composite choices. In fact, $\overline{K}$ can obtained with the function *duals*(K) of [12] that performs Reiter's hitting set algorithm over K, generating an element κ of $\overline{K}$ by picking an atomic choice (F, θ, k) from each element of K and inserting in κ the atomic choice $(F, \theta, 1-k)$. After this process is performed in all possible ways, inconsistent sets of atom choices are removed obtaining $\overline{K}$. Since the possible choices of the atomic choices are finite, so is $\overline{K}$. Finally, condition (a-3) holds since the union of ω_{K_1} with ω_{K_2} is equal to $\omega_{K_1 \cup K_2}$ for the definition of ω_K.

The corresponding measure μ is defined by $\mu(\omega_K) = P(K')$ where K' is a pairwise incompatible set of composite choices equivalent to K. $\langle W_\mathcal{P}, \Omega_\mathcal{P}, \mu\rangle$ is a finitely additive probability space according to Definition 6 because $\mu(\omega_{\{\emptyset\}}) = 1$, $\mu(\omega_K) \geq 0$ for all K and if $\omega_{K_1} \cap \omega_{K_2} = \emptyset$ and K_1' (K_2') is pairwise incompatible and equivalent to K_1 (K_2), then $K_1' \cup K_2'$ is pairwise incompatible and

$$\mu(\omega_{K_1} \cup \omega_{K_2}) = \sum_{\kappa \in K_1' \cup K_2'} P(\kappa) = \sum_{\kappa_1 \in K_1'} P(\kappa_1) + \sum_{\kappa_2 \in K_2'} P(\kappa_2) = \mu(\omega_{K_1}) + \mu(\omega_{K_2}).$$

Given a query q, a composite choice κ is an *explanation* for q if $\forall w \in \omega_\kappa : \ w \models q$. A set K of composite choices is *covering* wrt q if every world in which q is true belongs to ω_K

Definition 7. *For a probabilistic logic program $\mathcal{P}$, the probability of a ground atom q is given by $P(q) = \mu(\{w | w \in W_\mathcal{P}, w \models q\})$.*

If q has a finite set K of finite explanations such that K is covering then $\{w | w \in W_\mathcal{P} \wedge w \models q\} = \omega_K \in \Omega_T$ and we say that $P(q)$ is *finitely well-defined* for the distribution semantics. A program $\mathcal{P}$ is *finitely well-defined* if the probability of all ground atoms in the grounding of $\mathcal{P}$ is finitely well-defined.

Example 2. Consider the program of Example 1. The set $K = \{\kappa\}$ with $\kappa = \{(F_1, \{X/0\}, 1), (F_1, \{X/s(0)\}, 1)\}$ is a pairwise incompatible finite set of finite explanations that are covering for the query $p(s(0))$. Definition 7 therefore applies, and $P(p(s(0))) = P(\kappa) = a^2$

5 Infinite Covering Set of Explanations

In this section we go beyond [4] and we remove the requirement of the finiteness of the covering set of explanations and of each explanation for a query q.

Example 3. In Example 1, the query s has the pairwise incompatible covering set of explanations $K^s = \{\kappa_0^s, \kappa_1^s, \ldots\}$ with

$$\kappa_i^s = \{(F_2, \emptyset, 1), (F_1, \{X/0\}, 1), \ldots, (F_1, \{X/s^{i-1}(0)\}, 1), (F_1, \{X/s^i(0)\}, 0)\}$$

where $s^i(0)$ is the term where the functor s is applied i times to 0. So K^s is countable and infinite. A covering set of explanation for t is $K^t = \{\{(F_2, \emptyset, 0)\}, \kappa^t\}$ where κ^t is the infinite composite choice

$$\kappa^t = \{(F_2, \emptyset, 1), (F_1, \{X/0\}, 1), (F_1, \{X/s(0)\}, 1), \ldots\}$$

For a probabilistic logic program $\mathcal{P}$, we can define the probability measure $\mu : \Omega_\mathcal{P} \to [0, 1]$ where $\Omega_\mathcal{P}$ is defined as the set of sets of worlds identified by countable sets of countable composite choices: $\Omega_\mathcal{P} = \{\omega_K | K$ is a countable set of countable composite choices $\}$.

Lemma 3 *$\Omega_\mathcal{P}$ is a an σ-algebra over $W_\mathcal{P}$.*

Proof: (σ-1) is true as in the algebra case. To see that the complement ω_K^c of ω_K is in $\Omega_\mathcal{P}$, let us prove by induction that the dual $\overline{K}$ of K is a countable set of countable composite choices and then that $\omega_K^c = \omega_{\overline{K}}$. In the base case, if $K_1 = \{\kappa_1\}$, then we can obtain $\overline{K_1}$ by picking each atomic choice (F, θ, k) of κ_1 and inserting in $\overline{K_1}$ the composite choice $\{(F, \theta, 1-k)\}$. As there is a finite or countable number of atomic choices in κ_1, $\overline{K_1}$ is a finite or countable set of composite choices each with one atomic choice.

In the inductive case, assume that $K_{n-1} = \{\kappa_1, \ldots, \kappa_{n-1}\}$ and that $\overline{K_{n-1}}$ is a finite or countable set of composite choices. Let $K_n = K_{n-1} \cup \{\kappa_n\}$ and $\overline{K_{n-1}} = \{\kappa_1', \kappa_2', \ldots\}$. We can obtain $\overline{K_n}$ by picking each κ_i' and each atomic choice (F, θ, k) of κ_n. If $(F, \theta, k) \in \kappa_i'$, then discard κ_i', else if $(F, \theta, k') \in \kappa_i'$ with $k' \neq k$, insert κ_i' in $\overline{K_n}$. Otherwise generate the composite choice κ_i'' where $\kappa_i'' = \kappa_i' \cup \{(F, \theta, 1-k)\}$ and insert it in $\overline{K_n}$. Doing this for all atomic choices (F, θ, k) in κ_n generates a finite set of composite choices if κ_n is finite and a countable number if κ_n is countable. Doing this for all κ_i' we obtain that $\overline{K_n}$ is a countable union of countable sets which is a countable set [14, page 3]. $\omega_K^c = \omega_{\overline{K}}$ because all composite choices of $\overline{K}$ are incompatible with each world of ω_K, as they are incompatible with each composite choice of K. So $\omega_K^c \in \Omega_\mathcal{P}$. ($\sigma$-3) is true as in the algebra case. ♢

We can see K as $\lim_{n\to\infty} K_n$ where $K_n = \{\kappa_1, \ldots, \kappa_n\}$. Each K_n is a finite set of composite choices and we can compute an equivalent finite pairwise incompatible set of composite choices K_n'. For each K_n' we can compute the probability $P(K_n')$, noting that the probability of infinite composite choices is 0.

Now consider $\lim_{n\to\infty} P(K_n')$. We can see the $P(K_n')$s as the partial sums of a series. Moreover, it can be shown that $P(K_{n-1}') \leq P(K_n')$ so the series

has non-negative terms. Such a series converges if the sequence of partial sums is bounded from above [15, page 92]. Since $P(K'_n)$ is bounded by 1, the limit $\lim_{n\to\infty} P(K'_n)$ exists. So we can define measure μ as $\mu(\omega_K) = \lim_{n\to\infty} P(K'_n)$.

Theorem 4 *$\langle W_\mathcal{P}, \Omega_\mathcal{P}, \mu\rangle$ is a probability space according to Definition 4.*

Proof: (μ-1) and (μ-2) hold as for the finite case and for (μ-3) let $O = \{\omega_{L_1}, \omega_{L_2}, \ldots\}$ be a countable set of subsets of $\Omega_\mathcal{P}$ such that the ω_{L_i}s are pairwise disjoint. Let L'_i be the pairwise incompatible set equivalent to L_i and let $\mathcal{L}$ be $\bigcup_{i=1}^{\infty} L'_i$. Since the ω_{L_i}s are pairwise disjoint, then $\mathcal{L}$ is pairwise incompatible. $\Omega_\mathcal{P}$ is a σ-algebra, so $\mathcal{L}$ is countable. Let $\mathcal{L}$ be $\{\kappa_1, \kappa_2, \ldots\}$ and let K'_n be $\{\kappa_1, \ldots, \kappa_n\}$. Then $\mu(O) = \lim_{n\to\infty} P(K'_n) = \lim_{n\to\infty} \sum_{\kappa\in K'_n} P(\kappa) = \sum_{\kappa\in\mathcal{L}} P(\kappa)$. Since $\mathcal{L} = \bigcup_{i=1}^{\infty} L'_i$, by rearranging the terms in the last summation we get $\mu(O) = \sum_{\kappa\in\mathcal{L}} P(\kappa) = \sum_{n=1}^{\infty} P(L'_n) = \sum_{n=1}^{\infty} \mu(\omega_{L_n})$. ♢

For a probabilistic logic program $\mathcal{P}$, the probability of a ground atom q is again given by $P(q) = \mu(\{w|w \in W_\mathcal{P}, w \models q\})$. If q has a countable set K of explanations such that K is covering then $\{w|w \in W_\mathcal{P} \wedge w \models q\} = \omega_K \in \Omega_\mathcal{P}$ and we say that $P(q)$ is *well-defined* for the distribution semantics. A program $\mathcal{P}$ is well-defined if the probability of all ground atoms in the grounding of $\mathcal{P}$ is well-defined.

Example 4. Consider Example 3. Since the explanations in K^s are pairwise incompatible the probability of s can be computed as

$$P(s) = b(1-a) + ba(1-a) + ba^2(1-a) + \ldots = \frac{b(1-a)}{1-a} = b.$$

since the sum is a geometric series. K^t is also pairwise incompatible and $P(\kappa^t) = 0$ so $P(t) = 1 - b + 0 = 1 - b$ which is what we intuitively expect.

We now want to show that every program has countable set of countable explanations that is covering for each query. In the following, we consider only ground programs that however may be countably infinite, thus they can be the result of grounding a program with function symbols.

Given two sets of composite choices K_1 and K_2, define the conjunction $K_1 \otimes K_2$ of K_1 and K_2 as $K_1 \otimes K_2 = \{\kappa_1 \cup \kappa_2 | \kappa_1 \in K_1, \kappa_2 \in K_2, \mathit{consistent}(\kappa_1 \cup \kappa_2)\}$

Similarly to [7], we define parametrized interpretations and a $\mathit{IFPC}_\mathcal{P}$ operator. Differently from [7], here parametrized interpretations associate a set of composite choices instead of a Boolean formula to each atom.

Definition 8 (Parameterized two-valued interpretations). *A parameterized positive two-valued interpretation Tr of a ground probabilistic logic program $\mathcal{P}$ with and atoms $\mathcal{B}_\mathcal{P}$ is a set of pairs (a, K_a) with $a \in$ atoms and K_a a set of composite choices. A parameterized negative two-valued interpretation Fa of a ground probabilistic logic program P with atoms $\mathcal{B}_\mathcal{P}$ is a set of pairs $(a, K_{\neg a})$ with $a \in \mathcal{B}_\mathcal{P}$ and $K_{\neg a}$ a set of composite choices.*

Parametrized two-valued interpretations form a complete lattice where the partial order is defined as $I \leq J$ if $\forall (a, K_a) \in I, (a, L_a) \in J : \omega_{K_a} \subseteq \omega_{L_a}$. The

least upper bound and greatest lower bound always exist and are $lub(X) = \{(a, \bigcup_{(a,K_a)\in I, I\in X} K_a) | a \in \mathcal{B}_\mathcal{P}\}$ and $glb(X) = \{(a, \bigotimes_{(a,K_a)\in I, I\in X} K_a) | a \in \mathcal{B}_\mathcal{P}\}$. The top element $\top$ is $\{(a, \{\emptyset\}) | a \in \mathcal{B}_\mathcal{P}\}$ and the bottom element $\bot$ is $\{(a, \emptyset) | a \in \mathcal{B}_\mathcal{P}\}$.

Definition 9 (Parameterized three-valued interpretation). *A parameterized three-valued interpretation $\mathcal{I}$ of a ground probabilistic logic program $\mathcal{P}$ with atoms $\mathcal{B}_\mathcal{P}$ is a set of triples $(a, K_a, K_{\neg a})$ with $a \in \mathcal{B}_\mathcal{P}$ and K_a and $K_{\neg a}$ sets of composite choices.*

Parametrized three-valued interpretations form a complete lattice where the partial order is defined as $I \leq J$ if $\forall (a, K_a, K_{\neg a}) \in I, (a, L_a, L_{\neg a}) \in J : \omega_{K_a} \subseteq \omega_{L_a}$ and $\omega_{K_{\neg a}} \subseteq \omega_{L_{\neg a}}$. The least upper bound and greatest lower bound always exist and are $lub(X) = \{(a, \bigcup_{(a,K_a,K_{\neg a})\in I, I\in X} K_a, \bigcup_{(a,K_a,K_{\neg a})\in I, I\in X} K_{\neg a}) | a \in \mathcal{B}_\mathcal{P}\}$ and $glb(X) = \{(a, \bigotimes_{(a,K_a,K_{\neg a})\in I, I\in X} K_a, \bigotimes_{(a,K_a,K_{\neg a})\in I, I\in X} K_{\neg a}) | a \in \mathcal{B}_\mathcal{P}\}$. The top element $\top$ is $\{(a, \{\emptyset\}, \{\emptyset\}) | a \in \mathcal{B}_\mathcal{P}\}$, the bottom element $\bot$ is $\{(a, \emptyset, \emptyset) | a \in \mathcal{B}_\mathcal{P}\}$.

Definition 10. *For a ground program $\mathcal{P}$, a two-valued parametrized positive interpretation Tr with pairs (a, L_a), a two-valued parametrized negative interpretation Fa with pairs $(a, M_{\neg a})$ and a three-valued parametrized interpretation $\mathcal{I}$ with triples $(a, K_a, K_{\neg a})$, we define $OpTrueC^\mathcal{P}_\mathcal{I}(Tr) = \{(a, L'_a) | a \in \mathcal{B}_\mathcal{P}\}$ where*

$$L'_a = \begin{cases} \{\{(a, \emptyset, 1)\}\} & \text{if } a \in \mathcal{F} \\ \bigcup_{a \leftarrow b_1,\ldots,b_n,\neg c_1,\ldots,c_m \in \mathcal{R}} ((L_{b_1} \cup K_{b_1}) \otimes \ldots \\ \otimes (L_{b_n} \cup K_{b_n}) \otimes K_{\neg c_1} \otimes \ldots \otimes K_{\neg c_m}) & \text{if } a \in \mathcal{B}_\mathcal{P} \setminus \mathcal{F} \end{cases}$$

$OpFalseC^\mathcal{P}_\mathcal{I}(Fa) = \{(a, M'_a) | a \in \mathcal{B}_\mathcal{P}\}$ where

$$M'_{\neg a} = \begin{cases} \{\{(a, \emptyset, 0)\}\} & \text{if } a \in \mathcal{F} \\ \bigotimes_{a \leftarrow b_1,\ldots,b_n,\neg c_1,\ldots,c_m \in \mathcal{R}} ((M_{\neg b_1} \otimes K_{\neg b_1}) \cup \ldots \\ \cup (M_{\neg b_n} \otimes K_{\neg b_n}) \cup K_{c_1} \cup \ldots \cup K_{c_m}) & \text{if } a \in \mathcal{B}_\mathcal{P} \setminus \mathcal{F} \end{cases}$$

Proposition 5 *$OpTrueC^\mathcal{P}_\mathcal{I}$ and $OpFalseC^\mathcal{P}_\mathcal{I}$ are monotonic.*

Since $OpTrueC^\mathcal{P}_\mathcal{I}$ and $OpFalseC^\mathcal{P}_\mathcal{I}$ are monotonic, they have a least fixpoint and a greatest fixpoint.

Definition 11 (Iterated Fixed Point). *For a ground program $\mathcal{P}$, let $IFPC^\mathcal{P}$ be defined as $IFPC^\mathcal{P}(\mathcal{I}) = \{(a, K_a, K_{\neg a}) | (a, K_a) \in lfp(OpTrueC^\mathcal{P}_\mathcal{I}), (a, K_{\neg a}) \in lfp(OpFalseC^\mathcal{P}_\mathcal{I})\}$.*

Proposition 6 *$IFPC^\mathcal{P}$ is monotonic.*

So $IFPC^\mathcal{P}$ has a least fixpoint. Let $WFMC(\mathcal{P})$ denote $lfp(IFPC^\mathcal{P})$, and let δ the smallest ordinal such that $IFPC^\mathcal{P} \uparrow \delta = WFMC(\mathcal{P})$. We refer to δ as the *depth* of $\mathcal{P}$.

Theorem 7 *For a ground probabilistic logic program $\mathcal{P}$ with atoms $\mathcal{B}_\mathcal{P}$, let K_a^α and $K_{\neg a}^\alpha$ be the formulas associated with atom a in $IFPC^\mathcal{P} \uparrow \alpha$. For every atom a and total choice σ, there is an iteration α_0 such that for all $\alpha > \alpha_0$ we have:*

$$w_\sigma \in \omega_{K_a^\alpha} \leftrightarrow WFM(w_\sigma) \models a \quad w_\sigma \in \omega_{K_{\neg a}^\alpha} \leftrightarrow WFM(w_\sigma) \models \neg a$$

Theorem 8 *For a ground probabilistic logic program $\mathcal{P}$, let K_a^α and $K_{\neg a}^\alpha$ be the formulas associated with atom a in $IFPC^\mathcal{P} \uparrow \alpha$. For every atom a and every iteration α, K_a^α and $K_{\neg a}^\alpha$ are countable sets of countable composite choices.*

So every query for every program has a countable set of countable explanations that is covering and the probability measure is well defined. Moreover, since the program is sound, for all atoms a, $\omega_{K_a^\delta} = \omega^c_{K_{\neg a}^\delta}$ where δ is the depth of the program, as in each world a is either true or false.

5.1 Comparison with Sato and Kameya's Definition

Sato and Kameya [3] define the distribution semantics for definite programs. They build a probability measure on the sample space $W_\mathcal{P}$ from a collection of finite distributions. Let $\mathcal{F}$ be $\{F_1, F_2, \ldots\}$ and let X_i be a random variable associated to F_i whose domain is $\{0, 1\}$.

The finite distributions $P_\mathcal{P}^{(n)}(X_1 = k_1, \ldots, X_n = k_n)$ for $n \geq 1$ must be such that

$$\begin{cases} 0 \leq P_\mathcal{P}^{(n)}(X_1 = k_1, \ldots, X_n = k_n) \leq 1 \\ \sum_{k_1,\ldots,k_n} P_\mathcal{P}^{(n)}(X_1 = k_1, \ldots, X_n = k_n) = 1 \\ \sum_{k_{n+1}} P_\mathcal{P}^{(n+1)}(X_1 = k_1, \ldots, X_{n+1} = k_{n+1}) = P_\mathcal{P}^{(n)}(X_1 = k_1, \ldots, X_n = k_n) \end{cases} \tag{2}$$

The last equation is called the compatibility condition. It can be proved [16] from the compatibility condition that there exists a probability space $(W_\mathcal{P}, \Psi_\mathcal{P}, \eta)$ where η is a probability measure on $\Psi_\mathcal{P}$, the minimal σ-algebra containing open sets of $W_\mathcal{P}$ such that for any n,

$$\eta(X_1 = k_1, \ldots, X_n = k_n) = P_T^{(n)}(X_1 = k_1, \ldots, X_n = k_n). \tag{3}$$

$P_\mathcal{P}^{(n)}(X_1 = k_1, \ldots, X_n = k_n)$ is defined as $P_\mathcal{P}^{(n)}(X_1 = k_1, \ldots, X_n = k_n) = p_1 \ldots p_n$ where p_i is the annotation of alternative k_i in fact F_i. This definition clearly satisfies the properties in (2). $P_\mathcal{P}^{(n)}(X_1 = k_1, \ldots, X_n = k_n)$ is then extended to a probability measure over $\mathcal{B}_\mathcal{P}$.

We conjecture that this definition of the distribution semantics with function symbols coincides for definite programs with the one given above.

To show that the two definition coincide, we conjecture that $\Psi_\mathcal{P} = \Omega_T$. Moreover, $X_1 = k_1, \ldots, X_n = k_n$ is equivalent to the set of composite choices $K = \{\{(F_1, \emptyset, k_1), \ldots, (F_n, \emptyset, k_n)\}\}$ and $\mu(\omega_K)$ gives $p_1 \ldots p_n$ which satisfies equation (3).

6 Conclusions

We have presented a definition of the distribution semantics in terms of an iterated fixpoint operator that allowed us to prove that the semantics is well defined for all programs. The operator we have presented is also interesting from an inference point of view, as it can be used for forward inference similarly to [7].

References

1. Poole, D.: Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. New Gen. Comp. **11** (1993) 377–400
2. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: International Conference on Logic Programming. (1995) 715–729
3. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. J. Artif. Intell. Res. **15** (2001) 391–454
4. Poole, D.: The Independent Choice Logic for modelling multiple agents under uncertainty. Artif. Intell. **94** (1997) 7–56
5. Riguzzi, F., Swift, T.: Terminating evaluation of logic programs with finite three-valued models. ACM T. Comput. Log. **15** (2014) 32:1–32:38
6. Przymusinski, T.: Every logic program has a natural stratification and an iterated least fixed point model. In: ACM Conference on Principles of Database Systems. (1989) 11–21
7. Vlasselaer, J., Van den Broeck, G., Kimmig, A., Meert, W., De Raedt, L.: Anytime inference in probabilistic logic programs with Tp-compilation. In: Internation Joint Conference on Artificial Intelligence. (2015) 1852–1858
8. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM **38** (1991) 620–650
9. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Internation Joint Conference on Artificial Intelligence. (2007) 2462–2467
10. Kolmogorov, A.N.: Foundations of the Theory of Probability. Chelsea Publishing Company, New York (1950)
11. Srivastava, S.: A Course on Borel Sets. Graduate Texts in Mathematics. Springer (2013)
12. Poole, D.: Abducing through negation as failure: stable models within the independent choice logic. J. Logic Program. **44** (2000) 5–35
13. Poole, D.: Probabilistic Horn abduction and Bayesian networks. Artif. Intell. **64** (1993)
14. Cohn, P.: Basic Algebra: Groups, Rings, and Fields. Springer (2003)
15. Brannan, D.: A First Course in Mathematical Analysis. Cambridge University Press (2006)
16. Chow, Y., Teicher, H.: Probability Theory: Independence, Interchangeability, Martingales. Springer Texts in Statistics. Springer (2012)

A Proofs of Theorems

Proposition 5 *$OpTrueC^{\mathcal{P}}_{\mathcal{I}}$ and $OpFalseC^{\mathcal{P}}_{\mathcal{I}}$ are monotonic.*

Proof: Let us consider $OpTrueC^{\mathcal{P}}_{\mathcal{I}}$. We have to prove that if $Tr_1 \leq Tr_2$ then $OpTrueC^{\mathcal{P}}_{\mathcal{I}}(Tr_1) \leq OpTrueC^{\mathcal{P}}_{\mathcal{I}}(Tr_2)$. $Tr_1 \leq Tr_2$ means that $\forall (a, L_a) \in Tr_1, (a, M_a) \in Tr_2 : L_a \subseteq M_a$. Let (a, L'_a) be the elements of $OpTrueC^{\mathcal{P}}_{\mathcal{I}}(Tr_1)$ and (a, M'_a) the elements of $OpTrueC^{\mathcal{P}}_{\mathcal{I}}(Tr_2)$. We have to prove that $L'_a \subseteq M'_a$

If $a \in \mathcal{F}$ then $L'_a = M'_a = \{\{(a, \theta, 1)\}\}$. If $a \in \mathcal{B}_{\mathcal{P}} \setminus \mathcal{F}$, then L'_a and M'_a have the same structure. Since $\forall b \in \mathcal{B}_{\mathcal{P}} : L_b \subseteq M_b$, then $L'_a \subseteq M'_a$

We can prove similarly that $OpFalseC^{\mathcal{P}}_{\mathcal{I}}$ is monotonic. ♢

Proposition 6 *$IFPC^{\mathcal{P}}$ is monotonic.*

Proof: We have to prove that if $\mathcal{I}_1 \leq \mathcal{I}_2$ then $IFPC^{\mathcal{P}}(\mathcal{I}_1) \leq IFPC^{\mathcal{P}}(\mathcal{I}_2)$. $\mathcal{I}_1 \leq \mathcal{I}_2$ means that $\forall (a, L_a, L_{\neg a}) \in \mathcal{I}_1, (a, M_a, M_{\neg a}) \in \mathcal{I}_2 : L_a \subseteq M_a, L_{\neg a} \subseteq M_{\neg a}$. Let $(a, L'_a, L'_{\neg a})$ be the elements of $IFPC^{\mathcal{P}}(\mathcal{I}_1)$ and $(a, M'_a, M'_{\neg a})$ the elements of $IFPC^{\mathcal{P}}(\mathcal{I}_2)$. We have to prove that $L'_a \subseteq M'_a$ and $L'_{\neg a} \subseteq M'_{\neg a}$. This follows from the montonicity of $OpTrueC^{\mathcal{P}}_{\mathcal{I}_1}$ and $OpFalseC^{\mathcal{P}}_{\mathcal{I}_2}$ in $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively, which can be proved as in Proposition 5. ♢

Lemma 9 *For a ground probabilistic logic program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$, rules $\mathcal{R}$ and atoms $\mathcal{B}_{\mathcal{P}}$, let L^{α}_a be the formula associated with atom a in $OpTrueC^{\mathcal{P}}_{\mathcal{I}} \uparrow \alpha$. For every atom a, total choice σ and iteration α, we have:*

$$w_\sigma \in \omega_{L^{\alpha}_a} \rightarrow WFM(w_\sigma | \mathcal{I}) \models a$$

where $w_\sigma|\mathcal{I}$ is obtained by adding to w_σ the atoms a for which $(a, K_a, K_{\neg a}) \in \mathcal{I}$ and $w_\sigma \in K_a$ as facts and by removing all the rules with a in the head for which $(a, K_a, K_{\neg a}) \in \mathcal{I}$ and $w_\sigma \in K_{\neg a}$.

Proof: Let us prove the lemma by transfinite induction: let as assume the thesis for all $\beta < \alpha$ and let us prove it for α. If α is a successor ordinal, then it is easily verified for $a \in \mathcal{F}$. Otherwise assume $w_\sigma \in \omega_{L^{\alpha}_a}$ where

$$L^{\alpha}_a = \bigcup_{a \leftarrow b_1, \ldots, b_n, \neg c_1, \ldots, c_m \in \mathcal{R}} ((L^{\alpha-1}_{b_1} \cup K_{b_1}) \otimes \ldots \otimes (L^{\alpha-1}_{b_n} \cup K_{b_n}) \otimes K_{\neg c_1} \otimes \ldots \otimes K_{\neg c_m})$$

This means that there is rule $a \leftarrow b_1, \ldots, b_n, \neg c_1, \ldots, c_m \in \mathcal{R}$ such that $w_\sigma \in \omega_{L^{\alpha-1}_{b_i} \cup K_{b_i}}$ for $i = 1, \ldots, n$ and $w_\sigma \in \omega_{K_{\neg c_j}}$ for $j = 1 \ldots, m$. By the inductive assumption and because of how $w_\sigma|\mathcal{I}$ is built then $WFM(w_\sigma|\mathcal{I}) \models b_i$ and $WFM(w_\sigma|\mathcal{I}) \models \neg c_j$ so $WFM(w_\sigma|\mathcal{I}) \models a$.

If α is a limit ordinal, then

$$L^{\alpha}_a = lub(\{L^{\beta}_a | \beta < \alpha\}) = \bigcup_{\beta < \alpha} L^{\beta}_a$$

If $w_\sigma \in \omega_{L^{\alpha}_a}$ then there must exist a $\beta < \alpha$ such that $w_\sigma \in \omega_{L^{\beta}_a}$. By the inductive assumption the hypothesis holds. ♢

Lemma 10 *For a ground probabilistic logic program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$, rules $\mathcal{R}$ and atoms $\mathcal{B}_\mathcal{P}$, let $M^\alpha_{\neg a}$ be the set of composite choices associated with atom a in $OpFalseC^\mathcal{P}_\mathcal{I} \downarrow \alpha$. For every atom a, total choice σ and iteration α, we have:*

$$w_\sigma \in \omega_{M_{\neg a}} \rightarrow WFM(w_\sigma|\mathcal{I}) \models \neg a$$

where $w_\sigma|\mathcal{I}$ is built as in Lemma 9.

Proof: Similar to the proof of Theorem Lemma 9. ♢

Lemma 11 *For a ground probabilistic logic program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$, rules $\mathcal{R}$ and atoms $\mathcal{B}_\mathcal{P}$, let K^α_a and $K^\alpha_{\neg a}$ be the formulas associated with atom a in $IFPC^\mathcal{P} \uparrow \alpha$. For every atom a, total choice σ and iteration α, we have:*

$$w_\sigma \in \omega_{K^\alpha_a} \rightarrow WFM(w_\sigma) \models a \quad (4)$$

$$w_\sigma \in \omega_{K^\alpha_{\neg a}} \rightarrow WFM(w_\sigma) \models \neg a \quad (5)$$

Proof: Let us first prove that for all α, $WFM(w_\sigma) = WFM(w_\sigma|IFPC^\mathcal{P} \uparrow \alpha)$. We can prove it by transfinite induction. Consider the case of α a successor ordinal. Consider an atom b. If $w_\sigma \notin \omega_{K^\alpha_b}$ and $w_\sigma \notin \omega_{K^\alpha_{\neg b}}$ then the rules for b in w_σ and $w_\sigma|IFPC^\mathcal{P} \uparrow \alpha$ are the same. If $w_\sigma \in \omega_{K^\alpha_b}$ then b is a fact in $w_\sigma|IFPC^\mathcal{P} \uparrow \alpha$ but, according to Lemma 9, $WFM(w_\sigma|IFPC^\mathcal{P} \uparrow (\alpha-1)) \models b$. For the inductive hypothesis $WFM(w_\sigma) \models b$ so b has the same truth value in $WFM(w_\sigma)$ and $WFM(w_\sigma|IFPC^\mathcal{P} \uparrow \alpha)$. Similarly, if $w_\sigma \in \omega_{K^\alpha_{\neg b}}$, then $WFM(w_\sigma) \models \neg b$ and b has the same truth value in $WFM(w_\sigma)$ and $WFM(w_\sigma|IFPC^\mathcal{P} \uparrow \alpha)$. So overall $WFM(w_\sigma) = WFM(w_\sigma|IFPC^\mathcal{P} \uparrow \alpha)$.

If α is a limit ordinal, then $K^\alpha_b = \bigcup_{\beta<\alpha} K^\beta_b$ and $K^\alpha_{\neg b} = \bigcup_{\beta<\alpha} K^\beta_b$. So if $w_\sigma \in \omega_{K^\alpha_b}$ there is a β such $w_\sigma \in \omega_{K^\beta_b}$ and for the inductive hypothesis $WFM(w_\sigma) \models b$ so b has the same truth value in $WFM(w_\sigma)$ and $WFM(w_\sigma|IFPC^\mathcal{P} \uparrow \alpha)$. Similarly if $w_\sigma \in \omega_{K^\alpha_{\neg b}}$.

We can now prove the lemma by transfinite induction. Consider the case of α a successor ordinal. Since $(a, K^\alpha_a) \in lfp(OpTrueC^\mathcal{P}_{IFPC\uparrow(\alpha-1)})$, by Lemma 9

$$w_\sigma \in \omega_{K^\alpha_a} \rightarrow WFM(w_\sigma|IFPC^\mathcal{P} \uparrow (\alpha-1)) \models a$$

Since $WFM(w_\sigma|IFPC^\mathcal{P} \uparrow (\alpha-1)) = WFM(w_\sigma)$, (4) is proved.

Since $(a, K^\alpha_{\neg a}) \in gfp(OpFalseC^\mathcal{P}_{IFPC^\mathcal{P}\uparrow(\alpha-1)})$, by Lemma 10

$$w_\sigma \in \omega_{K^\alpha_{\neg a}} \rightarrow WFM(w_\sigma|IFPC^\mathcal{P} \uparrow (\alpha-1)) \models \neg a$$

Since $WFM(w_\sigma|IFPC^\mathcal{P} \uparrow (\alpha-1)) = WFM(w_\sigma)$, (5) is proved.

If α is a limit ordinal, $K^\alpha_a = \bigcup_{\beta<\alpha} K^\beta_a$ and $K^\alpha_{\neg a} = \bigcup_{\beta<\alpha} K^\beta_a$. If $w_\sigma \in \omega_{K^\alpha_a}$ there is a β such that $w_\sigma \in \omega_{K^\alpha_b}$ and by the inductive hypothesis (4) is proved. Similarly for (5). ♢

Lemma 12 *For a ground probabilistic logic program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$, rules $\mathcal{R}$ and atoms $\mathcal{B}_\mathcal{P}$, let K_a^α and $K_{\neg a}^\alpha$ be the formulas associated with atom a in $IFPC^\mathcal{P} \uparrow \alpha$. For every atom a, total choice σ and iteration α, we have:*

$$a \in IFP^{w_\sigma} \uparrow \alpha \rightarrow w_\sigma \in K_a^\alpha$$

$$\neg a \in IFP^{w_\sigma} \uparrow \alpha \rightarrow w_\sigma \in K_{\neg a}^\alpha$$

Proof: Let us prove it by double transfinite induction. If α is a successor ordinal, assume that

$$a \in IFP^{w_\sigma} \uparrow (\alpha - 1) \rightarrow w_\sigma \in K_a^{\alpha-1}$$

$$\neg a \in IFP^{w_\sigma} \uparrow (\alpha - 1) \rightarrow w_\sigma \in K_{\neg a}^{\alpha-1}$$

Let us perform transfinite induction on the iterations of $OpTrue^\mathcal{P}_{IFPC^\mathcal{P}\uparrow(\alpha-1)}$. Let us consider a successor ordinal δ: assume that

$$a \in OpTrue^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \uparrow (\delta - 1) \rightarrow w_\sigma \in L_a^{\delta-1}$$

$$\neg a \in OpFalse^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \downarrow (\delta - 1) \rightarrow w_\sigma \in M_{\neg a}^{\delta-1}$$

and prove that

$$a \in OpTrue^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \uparrow \delta \rightarrow w_\sigma \in L_a^{\delta}$$

$$\neg a \in OpFalse^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \downarrow \delta \rightarrow w_\sigma \in M_{\neg a}^{\delta}$$

Consider a. If $a \in \mathcal{F}$ then it is easily proved.

For other atoms a, $a \in OpTrue^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \uparrow \delta$ means that there is a rule $a \leftarrow b_1, \ldots, b_n, \neg c_1, \ldots, c_m$ such that for all $i = 1, \ldots, n$ $b_i \in OpTrue^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \uparrow (\delta - 1)$ and for all $j = 1, \ldots, m$ $\neg c_j \in IFP^{w_\sigma} \uparrow (\alpha - 1)$. For the inductive hypothesis $\forall i : w_\sigma \in L_{b_i}^{\delta-1} \vee w_\sigma \in K_{b_i}^{\alpha-1}$ and $\forall j : w_\sigma \in K_{\neg c_j}^{\alpha-1}$ so, for the definition of $OpTrue^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)}$, $w_\sigma \in L_a^\delta$. Analogously for $\neg a$.

If δ is a limit ordinal, then $L_a^\delta = \bigcup_{\mu<\delta} L_a^\mu$ and $M_{\neg a}^\delta = \bigotimes_{\mu<\delta} M_{\neg a}^\mu$. For the inductive hypothesis for all $\mu < \delta$

$$a \in OpTrue^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \uparrow \mu \rightarrow w_\sigma \in L_a^{\mu}$$

$$\neg a \in OpFalse^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \downarrow \mu \rightarrow w_\sigma \in M_{\neg a}^{\mu}$$

If $a \in OpTrue^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \uparrow \delta$, then there exists a $\mu < \delta$ such that $a \in OpTrue^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \uparrow \mu$. For the inductive hypothesis, $w_\sigma \in L_a^\delta$.

If $\neg a \in OpFalse^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \downarrow \delta$, then, for all $\mu < \delta$, $\neg a \in OpFalse^{w_\sigma}_{IFP^{w_\sigma}\uparrow(\alpha-1)} \downarrow \mu$. For the inductive hypothesis, $w_\sigma \in M_a^\delta$.

Consider a limit α. Then $K_a^\alpha = \bigcup_{\beta<\alpha} K_a^\beta$ and $K_{\neg a}^\alpha = \bigcup_{\beta<\alpha} K_{\neg a}^\beta$. The inductive hypothesis is

$$a \in IFP^{w_\sigma} \uparrow \beta \rightarrow w_\sigma \in K_a^{\beta}$$

$$\neg a \in IFP^{w_\sigma} \uparrow \beta \rightarrow w_\sigma \in K_{\neg a}^{\beta}$$

If $a \in IFP^{w_\sigma} \uparrow \alpha$, then there exists a $\beta < \alpha$ such that $a \in IFP^{w_\sigma} \uparrow \beta$. For the inductive hypothesis $w_\sigma \in K_a^\beta$ so $w_\sigma \in K_a^\alpha$. Similarly for $\neg a$. $\diamondsuit$

Theorem 7 *For a ground probabilistic logic program $\mathcal{P}$ with atoms $\mathcal{B}_\mathcal{P}$, let K_a^α and $K_{\neg a}^\alpha$ be the formulas associated with atom a in $IFPC^\mathcal{P} \uparrow \alpha$. For every atom a and total choice σ, there is an iteration α_0 such that for all $\alpha > \alpha_0$ we have:*

$$w_\sigma \in \omega_{K_a^\alpha} \leftrightarrow WFM(w_\sigma) \models a \quad w_\sigma \in \omega_{K_{\neg a}^\alpha} \leftrightarrow WFM(w_\sigma) \models \neg a$$

Proof: The $\rightarrow$ direction is Lemma 11. In the other direction, $WFM(w_\sigma) \models a$ implies $\exists \alpha_0 \forall \alpha \geq \alpha_0 : IFP_{w_\sigma} \uparrow \alpha \models a$. For Lemma 12, $w_\sigma \in \omega_{K_a^\alpha}$. $WFM(w_\sigma) \models \neg a$ implies $\exists \alpha_0 \forall \alpha \geq \alpha_0 : IFP^{w_\sigma} \uparrow \alpha \models \neg a$. For Lemma 12, $w_\sigma \in \omega_{K_{\neg a}^\alpha}$. $\diamondsuit$

Theorem 8 *For a ground probabilistic logic program $\mathcal{P}$, let K_a^α and $K_{\neg a}^\alpha$ be the formulas associated with atom a in $IFPC^\mathcal{P} \uparrow \alpha$. For every atom a and every iteration α, K_a^α and $K_{\neg a}^\alpha$ are countable sets of countable composite choices.*

Proof: It can be proved by observing that each iteration of $OpTrueC^\mathcal{P}_{IFPC^\mathcal{P}\uparrow\beta}$ and $OpFalseC^\mathcal{P}_{IFPC^\mathcal{P}\uparrow\beta}$ generates countable sets of countable explanations since the set of rules is countable. $\diamondsuit$

Probabilistic Abductive Logic Programming using Dirichlet Priors

Calin Rares Turliuc[1], Luke Dickens[2], Alessandra Russo[1], and Krysia Broda[1]

[1] Department of Computing, Imperial College London, United Kingdom
[2] Department of Information Studies, University College London
{ct1810, a.russo, k.broda}@imperial.ac.uk, l.dickens@ucl.ac.uk

Abstract. Probabilistic logic programming has traditionally focused on languages where probabilities or weights are specified or inferred directly, rather than through Bayesian priors. To address this limitation, we propose a probabilistic logic programming language that bridges the gap between logical and probabilistic inference in categorical models with Dirichlet priors. The language is described in terms of its general plate model, syntax, semantics and the relation between the three. A prototype implementation is evaluated on two case studies: latent Dirichlet allocation (LDA) on synthetic data, where we compare it with collapsed Gibbs sampling, and repeated insertion model (RIM) on real data. Universal probabilistic programming is not always scalable beyond toy examples on some models. However, our promising results show that the inference yields similar results to state-of-the-art solutions reported in the literature, produced with model-specific implementations.

Keywords: probabilistic programming, Bayesian inference, abductive logic programming, latent Dirichlet allocation, repeated insertion model

1 Introduction

Probabilistic programming is an area of research that aims to generalize inference in probabilistic models specified as inputs to a programming language. In this context, evaluating the program corresponds to prediction with or inference on the described model. A wide range of probabilistic programming languages (PPLs) have been developed, based on different programming languages and expressing a variety of classes of probabilistic models. Examples of PPLs include Church [8], Anglican [18], BUGS [15], Stan [22] and Figaro [19].[1] While some PPLs, such as Church, typically enrich a functional programming language with exchangeable random primitives, there also exist logic based PPLs that add probabilistic annotations or primitives to a logical encoding of the model. This encoding usually relates either to first-order logic, e.g. Alchemy[6], BLOG [17] or to logic programming PPLs, e.g. PRiSM [21], ProbLog [7].

[1] For a more comprehensive list cf. http://probabilistic-programming.org/.

Typical PPLs based on functional programming can express a wide range of probabilistic models, and inference is based on general sampling algorithms. Existing logic based PPLs mostly focus on discrete probabilistic models, and, generally, they do not consider Bayesian inference with prior distributions. For instance, Alchemy is a PPL which implements Markov logic, encoding a first order knowledge base into a Markov random field. Here, uncertainty is expressed by weights on the logical formulae and one cannot specify prior distributions on the weights. ProbLog is a PPL that primarily targets the inference of conditional probabilities and the most probable explanation (maximum likelihood solution) and it does not feature the specification of prior distributions. PRiSM is a PPL which introduces conjugate Dirichlet priors over categorical distributions; however, it is limited to probabilistic models described at the abductive level by non-overlapping explanations, such as hidden Markov models and probabilistic context-free grammars.

These observations motivate our present paper: we develop a logic programming based PPL specialized on probabilistic models involving categorical variables with conjugate Dirichlet priors that can be encoded as abductive logic programs with overlapping explanations. The programs evaluated by our PPL are abductive logic programs [11] enriched with probabilistic definitions and inference queries. We consider as case studies the latent Dirichlet allocation (LDA, [2]) and the repeated insertion model (RIM, [5]).

The contributions of this paper are:

- the design of `peircebayes`, a logic programming based PPL for inference in discrete models with categorical variables and Dirichlet priors.
- the description of the class of probabilistic models that can be expressed in the PPL, and their relation to the language.
- a prototype implementation of the language. For probabilistic inference, we adapt the Gibbs sampling algorithm described in [10].
- the formulation of RIM [5] as a probabilistic program.
- the evaluation of the PPL on an LDA task with synthetic data and on a RIM task with real data.

The rest of the paper is organized as follows. In Section 2 we describe the class of probabilistic models supported by our PPL. Section 3 explains the key features of the syntax and semantics of the PPL. We present the results of two experiments with our PPL in Section 4. Finally, in Section 5 we relate our PPL to other PPLs and methods, and we conclude.

2 The Probabilistic Model

This section introduces `peircebayes`[2], referred to in the rest of the paper as PB, a probabilistic logic programming language designed for inference in a subclass of the class of models called "propositional logic-based probabilistic models", described in [10].

[2] Named, in Church style, after Charles Sanders Peirce, the father of logical abduction, and Thomas Bayes, the father of Bayesian reasoning. Pronounced ['pɜrs'beɪz].

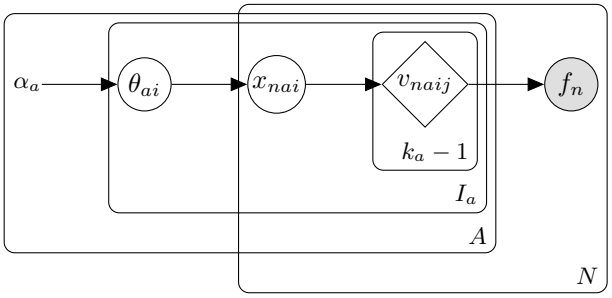


Fig. 1. The PB plate model. Unbounded nodes are constants, circle nodes are latent variables, shaded nodes are observed variables, diamond nodes are deterministic variables. A, I_a, N, and k_a are positive integers, $k_a \geq 2$.

The general plate notation [4] of the models expressible in PB is given in Figure 1. The plate model encodes the following (joint) probability distribution:

$$P(f, v, x, \theta; \alpha) = \left(\prod_{a=1}^{A} \prod_{i=1}^{I_a} P(\theta_{ai}; \alpha_a) \left(\prod_{n=1}^{N} P(x_{nai}|\theta_{ai}) P(v_{nai*}|x_{nai}) \right) \right) \prod_{n=1}^{N} P(f_n|v_{n*}) \quad (1)$$

We use $*$ to denote the set of variables obtained by iterating over the missing indexes, e.g. v_{nai*} is the set of all the variables v_{naij}, for $j = 1, \ldots, k_a - 1$, and v_{n*} is the set of all the variables v_{naij}, for $a = 1, \ldots, A$, $i = 1, \ldots, I_a$, $j = 1, \ldots, k_a - 1$. Unindexed variables are implicitly such sets, e.g. $x = x_*$.

In the model, each α_a, for $a = 1, \ldots, A$, is a vector of finite length $k_a \geq 2$ of positive real numbers. Each α_a may have a different length, and it represents the parameters of a Dirichlet distribution. From each such distribution I_a samples are drawn, i.e.:

$$\theta_{ai} \sim \text{Dirichlet}(\alpha_a) \qquad a = 1, \ldots, A \ , \ i = 1, \ldots, I_a$$

The samples θ are parameters of categorical distributions. Sampling N times from the latter yields:

$$x_{nai} \sim \text{Categorical}(\theta_{ai}) \qquad a = 1, \ldots, A \ , \ i = 1, \ldots, I_a \ , \ n = 1, \ldots, N$$

Each $x_{nai} \in \{1, \ldots, k_a\}$ is encoded, similarly to [20], as a set of propositional variables $v_{naij} \in \{0, 1\}$, for $j = 1, \ldots, k_a - 1$, in the following manner:

$$P(v_{nai*}|x_{nai} = l) = \begin{cases} \overline{v_{nai1}} \ldots \overline{v_{nail-1}} v_{nail} & , \text{ if } l < k_a \\ \overline{v_{nai1}} \ldots \overline{v_{nail-1}} & , \text{ if } l = k_a \end{cases}$$

where $\overline{v}$ denote boolean negation. Finally, the observed variables of the model, $f_n \in \{0, 1\}$, represent the output of boolean functions of v, such that:

$$P(f_n|v_{n*}) = [f_n = \text{Bool}_n(v_{n*})] \qquad n = 1, \ldots, N$$

$\text{Bool}_n(v)$ denotes an arbitrary boolean function of variables v, and $[i = j]$ is the Kronecker delta function δ_{ij}. The observed value for each f_n is 1 (or true) as we will explain in the following paragraph.

Inference in PB can be described in direct relation to a general schema of probabilistic inference, i.e. the characterization of $P(\theta|\Delta;\alpha)$, where θ are parameters of interest, α are constants (hyper-parameters) and Δ is some observed data. In PB, the parameters and the hyper-parameters correspond to θ and α, respectively. The observed data is captured by f and is assumed to be a set of N data points or observations. By convention, the realization $f_n = 1$ ensures that the n-th observation is included in the model, and, as such, we assume this is always the case. Furthermore, f_n is independent of the other observations given x (since x determines v), as implied by the joint distribution in Equation 1.

The various ways in which a data point can be generated, as well as the distributions involved in this process, are encoded through the boolean function $\text{Bool}_n(v_{n*})$ corresponding to the n-th data point. It is important to note that the data Δ can take any finite number of values, and $\text{Bool}_n(v_{n*})$ encodes the process of generating a single realization thereof.

Example. We illustrate the encoding of a popular probabilistic model for topic modelling, the latent Dirichlet allocation (LDA) [2] as a PB model. This will also serve as a running example throughout Section 3. LDA can be summarized as follows: given a corpus of D documents, each document is a list of tokens, the set of all tokens in the corpus is the vocabulary, with size V and assume there exist T topics. There are two sets of categorical distributions: D distributions over T categories, each distribution indexed μ_d, and T distributions over V categories, each distribution indexed ϕ_t. The words of a document d are produced independently by sampling a topic t from μ_d, then sampling a word from ϕ_t. Furthermore, each distribution in μ is sampled using the same Dirichlet prior with parameters γ, and, similarly, each distribution in ϕ is sampled using β. Note that μ and ϕ correspond to the parameters θ in the general model, and γ and β correspond to α. Assume that there is a corpus with 3 documents, 2 topics and a vocabulary of 4 words. The plate notation of the PB model of LDA is given in Figure 2.

Let the first data point be the observation of the second word of the vocabulary in document 3. Then the associated boolean function is:

$$\text{Bool}_1(v_{1*}) = v_{15}\overline{v_{111}}v_{112} + \overline{v_{15}}v_{121}v_{122}$$

The literals v_{15} and $\overline{v_{15}}$ denote the choice, in document 3, of topic 1 and 2, respectively, and the conjunctions $\overline{v_{111}}v_{112}$ and $\overline{v_{121}}v_{122}$ denote the choice of the second word from topic 1 and 2, respectively. Note that, in Figure 2, even though all possible edges between deterministic nodes and f_n are drawn, not all the variables must affect the probability of f_n, for instance the value of $\text{Bool}_1(v_{1*})$ doesn't depend on the value of v_{13}.

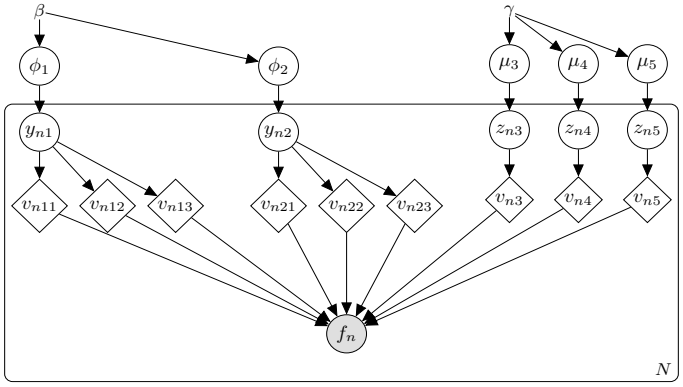


Fig. 2. The PB model for the LDA example.

3 Syntax and Semantics

Having established the semantics of the PB model, we proceed to describe the syntax and semantics of PB programs, and show how they relate to the probabilistic model.

A PB program is an abductive logic program [11] enhanced with probabilistic predicates. The abductive logic program encodes the generative story of the model, as well as the observed data. The most important probabilistic predicates are `pb_dirichlet` and `pb_plate`. The former provides a way to declare the probability distributions of the model, the latter is a query mechanism: it iterates through the data and computes all the possible ways it could have been generated according to the model, enabling the application of probabilistic inference algorithms.

The probabilistic predicate `pb_dirichlet` specifies a set of categorical distributions with the same Dirichlet prior, i.e. the elements on the outer plate indexed by a in Figure 1. Therefore, a set of such predicates express the whole outer plate. The syntax of the predicate is `pb_dirichlet(Alpha_a, Name, K_a, I_a)`. The first argument, `Alpha_a`, corresponds to α_a in the model, and can be either a list of k_a positive scalars specifying the parameters of the Dirichlet, or a positive scalar that specifies a symmetric prior. The second argument, `Name` is an atom that will be used as a functor when calling a predicate that represents a realization of a categorical random variable on the a-th plate. The third argument `K_a` corresponds to k_a, and `I_a` represents I_a, i.e. the number of categorical distributions having the same prior. The semantics of the predicate is that `Name` can be called in the program as a predicate, with the first argument denoting a category from $1, \ldots, k_a$, and the second argument a distribution from $1, \ldots, I_a$. In this paper, `Name(K_a, I_a)` is assumed to be a ground atom when called.

The probabilistic predicate `pb_plate(OuterQuery, Count, InnerQuery)` is the querying mechanism of PB. Informally, the first argument, `OuterQuery` is a usual Prolog query that iterates through the data. It must not call any predicates defined by `pb_dirichlet`. The argument `Count` is a positive integer that indicates that a particular observation is observed `Count` times. The final argu-

ment, `InnerQuery` is an abductive query that computes the explanation of the observations iterated in the `OuterQuery`. The formal semantics of `pb_plate` will be discussed after we introduce additional notation.

Example. Consider the LDA example from the previous section. Suppose we observe more data and we encode it in a PB program as:

```
observe(d(1),[(w(1),4),(w(4),2)]).
observe(d(2),[(w(3),1),(w(4),5)]).
observe(d(3),[(w(2),2)]).
```

Each `observe` fact encodes a document, indexed by an id using the first argument, and consisting of a bag-of-words in the second argument. The bag-of-words is a list of pairs: word index and its (positive) count per document. The next part of the PB program specifies the probability distributions as the following facts: `pb_dirichlet(1.0, mu, 2, 3)` and `pb_dirichlet(1.0, phi, 4, 2)`.

The `pb_plate` query iterates through document and word indexes and "explains" each such pair using the `generate` predicate. Note that `observe` and `generate` are not keywords, but descriptive conventional names.

```
pb_plate(
    [observe(d(Doc), TokenList), member((w(Token), Count), TokenList)],
    Count, [generate(Doc, Token)] ).

generate(Doc, Token) :- Topic in 1..2, mu(Topic, Doc), phi(Token, Topic).
```

Having described the core syntax of PB programs and its relation to the probabilistic model, we explain what is the result of executing a PB program and how probabilistic inference is performed to estimate $P(\theta|f, x, v; \alpha)$, or, in more typical applications, to produce an estimate $\hat{\theta}$.

In a traditional abductive logic programming setting [11,12,16], the result of evaluating a query is a list of abductive solutions, and an abductive solution is a list of abducibles. An abducible is a predicate that has no definition in the program. Some systems [20,23] represent an abductive solution as a pair of lists: a list of positive abducibles, i.e. abducibles that must be true, and a list of negative abducibles, i.e. abducibles that must be false. Since PB queries are abductive queries, an identical representation is obtained if the predicates defined by `pb_dirichlet` are parsed as annotated disjunctions [20].

Example. Consider the LDA example, more specifically the probabilistic predicate defining ϕ: `pb_dirichlet(1.0, phi, 4, 2)`. Let idx denote a positive integer such that `pa(Idx)` is a new abducible w.r.t. previously parsed `pb_dirichlet` predicates, i.e. the one defining μ. Abusing notation, $idx + incr$ is denoted by `Idx+incr`. The corresponding annotated disjunction is shown below:

```
phi(1,1) :- pa(Idx).
phi(2,1) :- \+pa(Idx), pa(Idx+1).
phi(3,1) :- \+pa(Idx), \+pa(Idx+1), pa(Idx+2).
phi(4,1) :- \+pa(Idx), \+pa(Idx+1), \+pa(Idx+2).
```

A similar annotated disjunction is produced for `phi(Token,2)`. Note that `pa(Idx)`, `pa(Idx+1)`, `pa(Idx+2)` represent v_{n11},v_{n12} and v_{n13} in Figure 2.

This encoding generates abductive representations of linear size in the number of categories of the distribution. In common LDA tasks, the size of the vocabulary of a corpus is frequently (much) more than 10.000 words, making it difficult to represent abductive solutions efficiently in a traditional way.

For the above reasons, PB uses a different representation of an abductive solution: a list of tuples (a, i, l), where a and i index the distribution as in the previous section, and l is a category $1 \leq l \leq k_a$. In the rest of the paper, the term "abductive solution" denotes this representation. The result of a PB query is a list of abductive solutions, and the result of calling `pb_plate` is a list of PB query results, one for each solution to `OuterQuery`. For the moment, consider programs with exactly one `pb_plate` definition. A generalization is presented once the necessary notation has been introduced.

Example. In the LDA program, the `OuterQuery` simply grounds `Doc` and `Token` in the order they are specified, e.g. the first grounding is $(1, 1)$, the second is $(1, 4)$, the third $(2, 3)$ etc. Assuming that square brackets represent lists, the result of `pb_plate` is shown below:

```
[   [ [(1,1,1),(2,1,1)], [(1,1,2),(2,2,1)] ],
    [ [(1,1,1),(2,1,4)], [(1,1,2),(2,2,4)] ],
    [ [(1,2,1),(2,1,3)], [(1,2,2),(2,2,3)] ],
    [ [(1,2,1),(2,1,4)], [(1,2,2),(2,2,4)] ],
    [ [(1,3,1),(2,1,2)], [(1,3,2),(2,2,2)] ] ]
```

The observation of word 2 in document 3 is the last element of the big list, and it can be produced using either topic 1 or 2, hence the two lists representing abductive solutions. The tuple $(1, 3, 1)$ means we choose topic 1 (last element) in document 3 (first two elements), and $(2, 1, 2)$ means we choose word 2 in topic 1 (with the same remarks).

Each result of a PB query is then parsed into a boolean formula which corresponds to $Bool_n(v_{n*})$ from the previous section. The key feature of PB is that it implicitly assumes that every result of a PB query on a `pb_plate` produces the same formula. This allows more concise query definitions, as well as improved time and memory performance, as a trade off with the user's expertise in PB. If the user were agnostic, she would write a `pb_plate` predicate for each data point (thus making the `OuterQuery` trivial). In future work we plan on investigating the automatic partition of the dataset in `pb_plate` definitions in an efficient way.

Example. The formula for the LDA example is $v_0v_1 + \overline{v_0}v_2$. Notice it is different from the one in Section 2, because the indexes have no semantic meaning w.r.t. the plate model, and the annotated disjunction is compiled w.r.t. the choices present in the same PB query, rather than the whole sample space.

The formula is compiled into a reduced ordered binary decision diagram (ROBDD, in the rest of the paper the RO attributes are implicit) [1,3], with the variables in ascending order according to their index. This means that the order of `pb_dirichlet` predicates matters, and that it should correspond to the sampling order in the generative story of the model. The BDD is the key data

structure that is used for probabilistic inference. The inference algorithm we use is an adaptation of Ishihata and Sato's Gibbs sampling for PLP models [10]. The algorithm is uncollapsed Gibbs sampling along two dimensions (θ and x).[3]

The difference between the original inference algorithm and the PB one is that instead of sampling from a Bernoulli, we sample from a multinomial with `Count` trials. We also sample all the (identical) BDDs for a `pb_plate` at once, using a single BDD, making sure to stop sampling a node when it isn't sampled in any of the implicit BDDs. Furthermore, sampling θ is followed by a re-parametrization such that the probabilities of the boolean variables in the BDD correspond to the new θ.

The generalization to multiple `pb_plate` predicates is straightforward: we sample each BDD, corresponding to one `pb_plate`, in turn, and all the samples update a common data structure representing x, and the probabilities of the boolean variables in each BDD are re-parametrized to adjust to the sampled θ.

In principle, it is possible to use the learned θ, or, to be more Bayesian, the posterior parameters of the Dirichlet α', to perform "forward" inference in a PB program: if we freeze θ, then the backward probability of the BDD yields the estimated parameter of a new observation. Otherwise, we sample x and θ using as priors α', record the backward probabilities of the BDD, then output the average thereof.

4 Evaluation

In this section we present experiments with PB[4]. No burn-in or lag was used in the experiments.

PB and collapsed Gibbs sampling (CGS) for LDA on synthetic data[5]. We run a variation of the experiment performed in [9,10]. A synthetic corpus is generated from an LDA model with parameters: 25 words in the vocabulary, 10 topics, 1000 documents, 100 words per document, and a symmetric prior on the mixture of topics μ, $\gamma = 1$. The topics used as ground truth specify uniform probabilities over 5 words, cf. [9,10]. We evaluate the convergence of PB and a traditional collapsed Gibbs sampling implementation[6]. The parameters are: $\beta = \gamma = 1$ as hyper-parameters, and we run 200 iterations of the samplers. The experiments are run 10 times over each corpus from a set of 10 identically sampled corpora, yielding 100 values of the log likelihoods per iteration. The average and 95% confidence interval (under a normal distribution) per iteration are shown in Figure 3. The experiment confirms the conclusions of the LDA experiment in [10]: both sampling algorithms converge, albeit PB converges slower

[3] The original Gibbs sampling for LDA [9] is collapsed Gibbs sampling along a number of dimensions equal to the number of words in the corpus.

[4] See supplementary materials for details on implementation and software availability (Appendix A).

[5] For details on likelihood formulation and comparison with the Church PPL, see supplementary materials (Appendices B and C).

[6] We use the `topicmodels` R package.

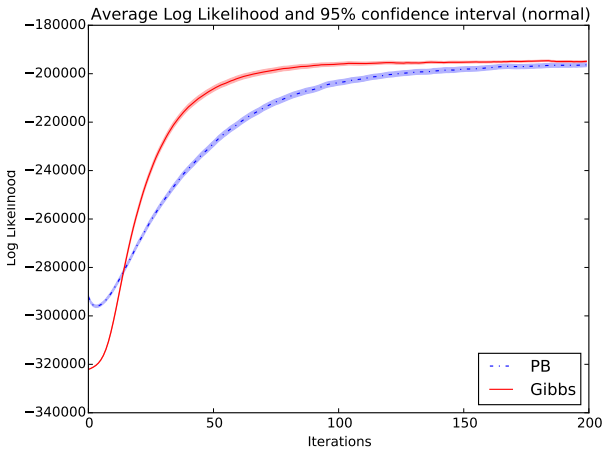


Fig. 3. Comparison between PB and collapsed Gibbs sampling on 10 sampled synthetic corpora (10 runs per corpus).

$\pi_1 = 0.144$	$\pi_2 = 0.191$	$\pi_3 = 0.153$	$\pi_4 = 0.185$	$\pi_5 = 0.187$	$\pi_6 = 0.138$
fatty tuna	fatty tuna	fatty tuna	fatty tuna	fatty tuna	fatty tuna
shrimp	tuna	sea urchin	sea urchin	tuna	tuna
sea eel	shrimp	salmon roe	salmon roe	shrimp	salmon roe
squid	squid	sea eel	shrimp	sea eel	shrimp
tuna	sea eel	tuna	sea eel	squid	squid
tuna roll	tuna roll	shrimp	squid	tuna roll	sea eel
salmon roe	egg	tuna roll	tuna roll	egg	tuna roll
sea urchin	cucumber roll	squid	tuna	salmon roe	sea urchin
egg	salmon roe	egg	egg	cucumber roll	egg
cucumber roll	sea urchin	cucumber roll	cucumber roll	sea urchin	cucumber roll

Table 1. Maximum likelihood preference profiles and mixture parameters on the Sushi dataset.

than CGS. However, we report different values for the log likelihood and note that PB takes 200 iterations rather than 100 to converge to a value that is close, under usual statistical assumptions, to the one produced by CGS.

PB for RIM on Sushi dataset. A repeated insertion model (RIM, [5]) provides a recursive and compact representation of K probability distributions, called preference profiles, over the set of all permutations of M items. This intuitively captures K different types of people with similar preferences. We evaluate a variant of the repeated insertion model in an experiment inspired by [14], on a dataset published in [13]. The data consists of 5000 permutations over $M = 10$ Sushi ingredients, each permutation expressing the preferences of a surveyed person. Following [14], we use $K = 6$ preference profiles, however we use the RIM rather than a Mallows model, and we train on the whole dataset. The parameters of the model are $50/K$ symmetric prior for the mixture of profiles, and 0.1 symmetric prior for all categorical distributions in all profiles. We run PB 10 times with 100 iterations and average the parameters. For each categorical

```
observe([5,0,3,4,6,9,8,1,7,2]).    observe([0,9,6,3,7,2,8,1,5,4]).
% ... 4998 'observe' facts ommited
pb_dirichlet(8.33333333333, pi, 6, 1).
pb_dirichlet(0.1, p2, 2, 6).       pb_dirichlet(0.1, p7, 7, 6).
pb_dirichlet(0.1, p3, 3, 6).       pb_dirichlet(0.1, p8, 8, 6).
pb_dirichlet(0.1, p4, 3, 6).       pb_dirichlet(0.1, p9, 9, 6).
pb_dirichlet(0.1, p5, 5, 6).       pb_dirichlet(0.1, p10, 10, 6).
pb_dirichlet(0.1, p6, 6, 6).
pb_plate( [observe(Sample)], 1,
    [generate([0,1,2,3,4,5,6,7,8,9], Sample)] ).
generate([H|T], Sample):-
    K in 1..6, pi(K, 1), generate(T, Sample, [H], 2, K).
generate([], Sample, Sample, _Idx, _K).
generate([ToIns|T], Sample, Ins, Idx, K) :-
    % insert next element at Pos yielding a new list Ins1
    append(_, [ToIns|Rest], Sample),
    insert_rim(Rest, ToIns, Ins, Pos, Ins1),
    % build prob predicate in Pred
    number_chars(Idx, LIdx), append(['p'], LIdx, LF),
    atom_chars(F, LF), Pred =.. [F, Pos, K],
    % call prob predicate and recurse
    pb_call(Pred), Idx1 is Idx+1,
    generate(T, Sample, Ins1, Idx1, K).
insert_rim([], ToIns, Ins, Pos, Ins1) :-
    append(Ins, [ToIns], Ins1), length(Ins1, Pos).
insert_rim([H|_T], ToIns, Ins, Pos, Ins1) :-
    nth1(Pos, Ins, H), nth1(Pos, Ins1, ToIns, Ins).
insert_rim([H|T] , ToIns, Ins, Pos, Ins1) :-
    \+member(H, Ins), insert_rim(T, ToIns, Ins, Pos, Ins1).
```

Table 2. PB program for a RIM with $K = 6$ preference profiles.

distribution in a profile, we select its maximum likelihood realization to build the corresponding maximum likelihood preference profile, shown in Table 1. The inference yields similar conclusions to [14]: there is a strong preference for fatty tuna, a strong dislike of cucumber roll and a strong positive correlation between salmon roe and sea urchin. We show the PB program used in Table 2, noting that `pb_call/1` is a special PB predicate that allows the evaluation of its argument as a predicate defined by `pb_dirichlet`. We are not aware of any other implementation of RIM in a PPL, therefore we briefly describe the program. The mixture of profiles is characterized by π, a set of K distributions, and for each profile there are $M - 1$ categorical distributions that specify the probabilities over the set of permutations of M elements. An observed permutation is produced by selecting a latent profile, then generating that permutation by consecutively inserting elements from an insertion order, e.g. $[0, 1, \ldots, 9]$, at the right position, according to the distributions in that profile. The right position is chosen using the `insert_rim` predicate, as naively generating all the possible permutations is intractable.

5 Related Work and Conclusions

In this paper, we introduced PB, a probabilistic logic programming language for categorical models with Dirichlet priors. The idea of PB was sparked by [10], which defines a similar class of probabilistic models and provides a Gibbs sampling algorithm for BDDs. However, BDDs are not a programming language, nor are they an intuitive representation for a non-expert. This paper bridges the gap between logical and probabilistic inference in the considered class of models, and addresses issues on representation of abductive solutions and inference on "syntactically" identical BDDs. Similarly to ProbLog [7], the pipeline of PB can be described as logical inference, followed by knowledge compilation, followed by probabilistic evaluation. Unlike ProbLog, the most difficult task in PB is probabilistic evaluation, rather than knowledge compilation, though for complex programs, PB could benefit from using more compact decision diagrams. In relation to Church [8] and many other related PPLs, PB is similar in that it uses a Turing-complete declarative language, but the set of probabilistic primitives available in PB is very restricted compared to Church. PB uses a different probabilistic model than Alchemy [6], and by using abductive logic programming instead of a first-order knowledge base, it can easily encode recursive generative models, such as RIM. Although in this paper we present well studied models, they can be easily adapted to include various constraints, e.g. seed words in LDA. In future work, we hope to explore more probabilistic models that fit the PB paradigm, and to design, implement, and compare efficient algorithms for generalized probabilistic inference.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. 27(6), 509–516 (Jun 1978), http://dx.doi.org/10.1109/TC.1978.1675141
2. Blei, D.M., Ng, A.Y., Jordan, M.I., Lafferty, J.: Latent dirichlet allocation. Journal of Machine Learning Research 3, 2003 (2003)
3. Bryant, R.: Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on C-35(8), 677–691 (Aug 1986)
4. Buntine, W.L.: Operations for learning with graphical models. J. Artif. Intell. Res. (JAIR) 2, 159–225 (1994), http://dx.doi.org/10.1613/jair.62
5. Doignon, J.P., Peke, A., Regenwetter, M.: The repeated insertion model for rankings: Missing link between two subset choice models. Psychometrika 69(1), 33–54 (2004), http://dx.doi.org/10.1007/BF02295838
6. Domingos, P., Kok, S., Poon, H., Richardson, M., Singla, P.: Unifying logical and statistical AI. In: Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1. pp. 2–7. AAAI'06, AAAI Press (2006), http://dl.acm.org/citation.cfm?id=1597538.1597540
7. Fierens, D., den Broeck, G.V., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., Raedt, L.D.: Inference and learning in probabilistic logic programs using weighted boolean formulas. CoRR abs/1304.6810 (2013), http://arxiv.org/abs/1304.6810

8. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. Uncertainty in Artificial Intelligence 2008 (2008), http://danroy.org/papers/church_GooManRoyBonTen-UAI-2008.pdf
9. Griffiths, T.L., Steyvers, M.: Finding scientific topics. Proceedings of the National Academy of Sciences 101(suppl 1), 5228–5235 (2004), http://www.pnas.org/content/101/suppl_1/5228.abstract
10. Ishihata, M., Sato, T.: Bayesian inference for statistical abduction using Markov chain Monte Carlo. In: Proceedings of the 3rd Asian Conference on Machine Learning, ACML 2011, Taoyuan, Taiwan, November 13-15, 2011. pp. 81–96 (2011), http://www.jmlr.org/proceedings/papers/v20/ishihata11/ishihata11.pdf
11. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming (1993)
12. Kakas, Antonis, C., Van Nuffelen, B., Denecker, M.: A-system : Problem solving through abduction. In: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence. vol. 1, pp. 591–596. Morgan Kaufmann Publishers, Inc (2001), http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=34862
13. Kamishima, T., Kazawa, H., Akaho, S.: Supervised ordering - an empirical survey. In: Data Mining, Fifth IEEE International Conference on. pp. 4 pp.– (Nov 2005)
14. Lu, T., Boutilier, C.: Effective sampling and learning for Mallows models with pairwise-preference data. Journal of Machine Learning Research 15, 3783–3829 (2014), http://jmlr.org/papers/v15/lu14a.html
15. Lunn, D., Spiegelhalter, D., Thomas, A., Best, N.: The BUGS project: Evolution, critique and future directions. Statistics in Medicine 28(25), 3049–3067 (2009), http://dx.doi.org/10.1002/sim.3680
16. Ma, J.: Abductive reasoning module for SICStus prolog (2012), http://www-dse.doc.ic.ac.uk/cgi-bin/moin.cgi/abduction
17. Milch, B., Marthi, B., Russell, S.: Blog: Relational modeling with unknown objects. In: ICML 2004 Workshop on Statistical Relational Learning and Its Connections. pp. 67–73 (2004)
18. Paige, B., Wood, F.: A compilation target for probabilistic programming languages. CoRR abs/1403.0504 (2014), http://arxiv.org/abs/1403.0504
19. Pfeffer, A.: Figaro: An object-oriented probabilistic programming language (2009)
20. Raedt, L.D., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M.M. (ed.) IJCAI. pp. 2462–2467 (2007), http://dblp.uni-trier.de/db/conf/ijcai/ijcai2007.html#RaedtKT07
21. Sato, T., Kameya, Y.: New advances in logic-based probabilistic modeling by prism. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.) Probabilistic Inductive Logic Programming, Lecture Notes in Computer Science, vol. 4911, pp. 118–155. Springer Berlin Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78652-8_5
22. Stan Development Team: Stan Modeling Language Users Guide and Reference Manual, Version 2.5.0 (2014), http://mc-stan.org/
23. Turliuc, C., Maimari, N., Russo, A., Broda, K.: On minimality and integrity constraints in probabilistic abduction. In: Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings. pp. 759–775 (2013), http://dx.doi.org/10.1007/978-3-642-45221-5_51

A Implementation

PB is implemented in YAP and Python (2.7), and is currently available as a command-line script. YAP is used to parse input files and produce files for probabilistic inference (e.g. solutions to each `pb_plate` query, information on the probability distributions). PyCUDD is used to compile ROBDDs and computationally intensive parts of the sampling algorithm are implemented in Cython. This prototype implementation and any additional files are released under a GNU General Public License (GPL3).

For more information and documentation see:
`http://raresct.github.io/peircebayes`
To access the source code see:
`http://www.github.com/raresct/peircebayes`
To reproduce the experiments, more concretely Figure 3 and Table 1, see:
`http://www.github.com/raresct/peircebayes_experiments`
On an Intel® Core™ i7-4710HQ CPU @ 2.50GHz ×8, the LDA experiment took: $\approx$ 265 minutes for PB, $\approx$ 4 minutes for CGS, and the RIM experiment took $\approx$ 10 minutes. Note that there is significant overhead for PB because we don't measure only sampling time, but also logical inference and knowledge compilation.

B A Note on the Joint Distribution of the PB model and Likelihood for LDA

The joint distribution of collapsed PB models is:

$$P(f, v, x; \alpha) = P(f|v)P(v|x)P(x; \alpha)$$

If x is the result of sampling the appropriate BDDs, then $P(f|v)P(v|x) = 1$, and the joint distribution reduces to $P(x; \alpha)$. This type of distribution has been well studied, cf. equations 2 and 3 in [9] for LDA, and in the case of PB models, it is:

$$P(x; \alpha) = \prod_{a=1}^{A} \left(\left(\frac{\Gamma(\sum_{l=1}^{k_a} \alpha_{al})}{\prod_{l=1}^{k_a} \Gamma(\alpha_{al})} \right)^{I_a} \prod_{i=1}^{I_a} \frac{\prod_{l=1}^{k_a} \Gamma(\sum_{n=1}^{N} [x_{nai} = l] + \alpha_{al})}{\Gamma(\sum_{n=1}^{N} x_{nai} + \sum_{l=1}^{k_a} \alpha_{al})} \right)$$

The likelihood for LDA is recovered by using the factors of the joint distribution involving only β, ϕ, y.

C PB and Church on LDA

In Section 4 we compare PB with CGS on a synthetic LDA task. We add to the comparison a much more expressive, universal PPL called Church [8]. The experimental setting differs from the LDA experiment in Section 4 in that we

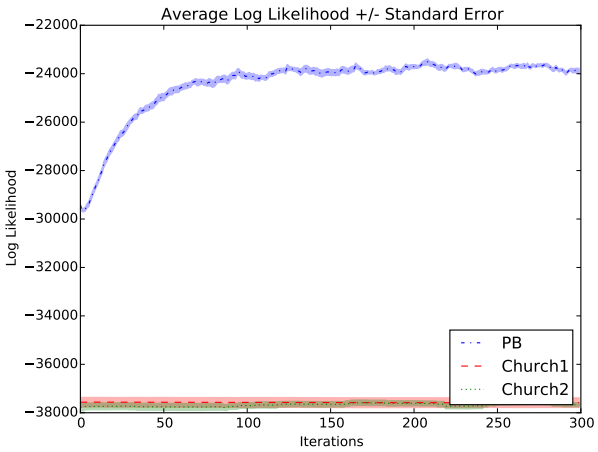


Fig. 4. Comparison between PB and Church on one sampled synthetic corpus (10 runs per corpus).

sample only one synthetic corpus of 100 documents. This is due to the fact that the Church implementation of LDA is slow. Furthermore, we take 300 samples (no lag, no burn-in for PB, 10 lag, no burn-in for Church). We use two implementations of LDA in Church[7], and report the results in Figure 4. Note that we use the uncollapsed likelihood for the Church models (which is more "optimistic" than the collapsed one), mainly due to the fact that we were unable to find an implementation of the $\log \Gamma$ function in Church.

Neither Church LDA programs seems to converge, while PB behaves consistently with the previous experiment. The average time per run is: ≈ 0.36 minutes for PB, ≈ 13.5 minutes for Church1 and ≈ 16.7 minutes for Church2.

[7] We use adaptations of the two programs shown here: `http://forestdb.org/models/lda.html` and run them with `webchurch` (`https://github.com/probmods/webchurch`) as command-line scripts.