Probabilistic logic programming in 2P-KT

Jason Dellaluce[1], Roberta Calegari[2] and Giovanni Ciatto[1]

[1]Dipartimento di Informatica – Scienza e Ingegneria (DISI), Alma Mater Studiorum—Università di Bologna, Italy
[2]Alma Mater Research Institute for Human-Centered Artificial Intelligence, Alma Mater Studiorum—Università di Bologna, Italy

Abstract
The work introduces an elastic and platform-agnostic approach to probabilistic logic programming aimed at linking this paradigm with modern mainstream programming platforms, thus widening its usability and portability (e.g. towards the JVM, Android, Python, and JavaScript platforms). We design our solution as an extension of the 2P-Kt symbolic AI ecosystem to inherit its multi-platform and multi-paradigm nature.

Keywords
probabilistic logic programming, symbolic AI, 2P-Kt

1. Introduction

Artificial Intelligence (AI) is progressively conquering the software industry to become one of the most pivotal fields, with a fast-paced evolution of challenges and requirements that existing technologies often fail to match. Accordingly, the increasing demand for transparent and pervasive intelligence is opening new horizons for logic programming (LP) and symbolic AI approaches [1, 2, 3]. However, logic-based approaches alone are often not suitable to be integrated with present-day planning and learning workflows, which natively deal with uncertainty and probabilistic decision-making [4, 5, 6].

Probabilistic logic programming (PLP) [7, 8] is a research field that investigates the combination of LP with the probability theory. There, theories may contain facts or rules enriched with probabilities, which may, in turn, be queried by the users to investigate not only which statements are true or not, but also under which probability. To support this behaviour, probabilistic solvers leverage ad-hoc resolution strategies explicitly taking probabilities into account. This makes them ideal to deal with uncertainty and the complex phenomena of the physical world. It is thus unsurprising that Bayesian and data-driven AI, other than cyber physical systems (CPS), are among the areas which would benefit the most from the development of robust and interoperable PLP technologies.

State-of-the-art PLP solutions [9, 10] have reached a considerable level of maturity and theoretical reach. Not only has *exact* probabilistic resolution been reified into actual programming

AIxIA 2021 Discussion Papers
jason.dellaluce@studio.unibo.it (J. Dellaluce); roberta.calegari@unibo.it (R. Calegari); giovanni.ciatto@unibo.it (G. Ciatto)
http://robertacalegari.apice.unibo.it (R. Calegari); https://about.me/gciatto (G. Ciatto)
0000-0003-3794-2942 (R. Calegari); 0000-0002-1841-8996 (G. Ciatto)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)
CEUR-WS.org/Vol-3078/paper-29.pdf

languages, but also approximate resolution, and learning of probabilities from data. However, existing technologies currently rely upon monolithic runtimes, often targeting single platforms or having inconvenient constraints and dependencies [11, 12]—limiting their interoperability and portability with mainstream programming platforms. This follows a general tendency of logic-based technologies, which are often constructed as technological silos – being so optimised for performance and correctness while being poorly interoperable among each other – targetting the LP community alone.

To overcome such tendency towards the creation of isolated monoliths, the notion of *logic ecosystem* [13] has recently been proposed. There, the authors argue that LP facilities – e.g. knowledge representation, unification, clauses indexing, resolution, etc. – should be made independently available to the widest possible audience—there including mainstream developers and logic programmers, and all major programming platforms. Most notably, LP facilities should not only be exploitable as stand-alone applications (e.g. Prolog interpreters) but also (and foremost) as libraries—thus enabling re-use at the mechanism level. In this perspective, logic ecosystems consist of *extensible* technological *frameworks* where single LP facilities can be incrementally constructed on top of the previous ones, other than used—either individually or composedly. Notably, the authors in [13] prose 2P-Kt as the technological reification of a logic ecosystem. Unfortunately, however, PLP is not among the LP facilities currently supported by 2P-Kt.

Accordingly, in this work we propose an extension of the 2P-Kt ecosystem aimed at supporting PLP via an ad-hoc implementation of the ProbLog language. The proposed implementation aims at overcoming the interoperability and portability issues of state-of-the-art PLP solutions. In fact, as part of the 2P-Kt ecosystem, our ProbLog implementation can be compiled/run on several strategic platforms, other than used as a library in multiple programming languages. Our solution provides PLP support on top of standard Prolog solvers. Hence, as a side contribution, we provide insights about how a ProbLog solver can be realised on top of Prolog's SLD(+NF) resolution principle.

It is worth highlighting how our current goal is to provide a usable and functioning PLP code base, initially supporting only the fundamental features, and aiming to be flexible for future growth. Outperforming existing solutions is not amongst our primary concerns. Conversely, we aim to open the horizons for wider adoption of LP and PLP, by favouring portability and by making it easier to exploit from outside the LP realm. In this regard, we describe a number of examples aimed at demonstrating the usability and portability of our PLP solution on multiple runtimes and programming platforms.

2. Background

A variety of research contributions exploring the field of PLP exist in the logic programming literature. Proposals often differ for their semantics or syntaxes, or for the way they perform probabilistic reasoning [14, 11, 9, 15].

Roughly speaking, semantics are concerned with endowing probabilistic programs with meaning. Sato's distribution semantics (DS) [16, 17] is one of the most prominent approaches for the combination of logic programming and probability theory. There, a probabilistic logic

program is interpreted as a concise description of many possible worlds, and the probabilities of queries are solved by summing up their probability in each possible world.

Languages adhering to the distribution semantics may in turn differ in how they represent clauses, and their probabilities. A successful approach in this context is LPAD (Logic Programs with Annotated Disjunctions), where clauses admit disjunctions of atoms in their heads, and each atom is labelled with a probability value. In other words, LPAD is a special notation supporting the definition of non-binary probabilistic distributions over clauses and facts. However, in practice, probabilistic logic programs may support a certain *evidence* [18] to be provided via unannotated fact/rules which are known to be true, even though they may be defined over some probability distribution.

Finally, concerning probabilistic reasoning, PLP generally supports reasoning tasks, and each of them has been richly documented in the literature [19]. Broadly speaking, options range from exact to approximate—the former being more precise and computationally demanding, while the latter being more affordable at the price of lower precision. In the remainder of this paper we focus on exact methods only.

Along this line, a common strategy is to rely upon knowledge compilation [20] to make probabilistic reasoning efficient—i.e., by transforming logic formulæ into simpler (more tractable) forms. *Binary decision diagrams* (BDD) [21, 22] and their variants/extensions are commonly exploited to serve this purpose [23, 24].

2.1. State-of-the-art technologies for PLP

A number of programming languages follow the LPAD approach over the DS, there including ProbLog and `cplint`. They both rely on (some variant of) BDD to support probabilistic reasoning. Within the scope of this paper, we consider them as interesting solutions for PLP as they come with some actually usable technology. In the reminder of this section, we briefly analyse ProbLog and `cplint` from a technological perspective.

ProbLog. ProbLog [9] is a probabilistic programming language providing PLP support on top of Prolog. We appreciate the simplicity of the language and the high compatibility with traditional Prolog—hence why we target a ProbLog extension for 2P-Kt. ProbLog, in particular, leverages upon a number of aspects of Prolog's operation to attain PLP support. First, it relies on knowledge compilation of annotated facts into ordinary Prolog clauses. Then, it exploits Prolog's backtracking mechanism to enumerate the possible worlds in which a query is true. These are called 'explanations' in PLP's nomenclature, while they are ordinary solutions in the eyes of a Prolog solver. Finally, ProbLog attempts to iteratively build a BDD as part of the resolution process, in order to keep the problem of computing the probability of a query tractable. The Prolog solvers' dynamic KB are used as ancillary data stores in the meanwhile. Once all the possible worlds have been enumerated, the resulting BDD is fully navigated to efficiently compute the probability of the query.

Currently, the ProbLog project consists of a Python codebase, depending on a number of native libraries and tools—such as the YAP Prolog technology [25]. Such technological choices limit the portability of ProbLog outside the scope of the major desktop operative systems (e.g.

Windows, Linux, or Mac OS). Notably, this issue is mitigated by the existence of a publicly-available Web application letting users experiment ProbLog from their browsers. In any case, to the best of our understanding of the ProbLog's documentation and source code, ProbLog is mainly intended as a stand-alone command-line application and interpreter, and its usage as a library is not explicitly supported.

`cplint`. The `cplint` system (CPLogic INTerpreter) [10] applies knowledge compilation to logic programs annotated à la CP-Logic [26]. Notably, it compiles probabilistic clauses into Multivalued Decision Diagrams (MDDs) [27], an extension of BDDs. Thus, differently from ProbLog, the random variables corresponding to logic clauses can be multi-valued. Furthermore, `cplint`'s probabilistic programs support negated atoms.

`cplint` leverages upon a Prolog meta-interpreter to solve probabilistic queries. Similarly to ProbLog, it keeps track of the solutions encountered during resolution, while simultaneously building a MDD aimed at leter being able to draw probabilities.

Currently, the `cplint` project consists of a Prolog codebase targetting the SWI-Prolog [28] platform. Such technological choices limit the portability of `cplint` on platforms for which SWI-Prolog is not available, or platforms that are poorly interoperable with (SWI-)Prolog—e.g. Android, the JVM or iOS. Notably, this issue is mitigated by the existence of a publicly-available Web application letting users experiment `cplint` from their browsers. In any case, to the best of our understanding of its documentation and source code, `cplint` is mainly intended as a stand-alone command-line application and interpreter, or as a Prolog library.

2.2. Logic Ecosystems and 2P-Kt

The current practice of logic-based technologies (LBT) follows a tendency where software contributions are constructed as extensions or on top of the Prolog language, often on native (i.e. based on C or C++) technologies. Such a tendency has pushed the LP community towards a situation where tools consist of poorly interoperable technological silos, where: *(i)* logic facilities (e.g. unification; clauses storage, indexing, or retrieval; resolution, etc.) are not adequately separated, and can only be exploited by means of Prolog, *(ii)* usage of logic facilities must step through a stand-alone application (commonly, either graphical or command-line), as they are not available "as a library" to other programming platforms *(iii)* the portability of LBT technologies is constrained on the platforms the underlying Prolog system supports.

To overcome such issues the 2P-Kt technology has been recently proposed in [13], along with the notion of logic *ecosystem*. There 2P-Kt is considered as an ecosystem of loosely coupled *modules*, each one dedicated to a single logic facility. Hence, overall, it consists of a collection of logic facilities, exposed to the developers as *multi-platform* libraries—and, possibly, as stand-alone applications as well. There, multi-platform support aims at letting mainstream programming platforms benefit from the sole logic facilities they need, natively—and without having to interact with a full fledged Prolog system.

Arguably, multi-platform support is fundamental to let researchers and practitioners from the many branches of computer science and artificial intelligence benefit from LBT. Along this line, we believe logic facilities – such as probabilistic resolution – should be exploitable on mainstream programming platforms and languages – e.g. JVM, Python, JavaScript, etc. – to ease

the exploitation of LP for the niches by which those platforms and languages are used the most. On the long run, for instance, we hope that bringing LP on Python will ease its hybridisation with data science, while bringing it on JavaScript will ease its hybridisation with the Web, and so on.

Accordingly, 2P-Kt currently explicitly targets the Kotlin, Android, JVM, and JavaScript platforms, while other platforms – such as iOS and Python – are going to be supported soon, thanks to the multi-platform programming facilities offered by Kotlin[1]. Of course, we acknowledge that different languages and platforms may follow different conventions and paradigms. Hence, multi-platform must not be realised via mere cross-compilation on several platforms, but rather ad-hoc software layers should be provided to harmonise LP to the target platforms, at the paradigm level (cf. [29]).

2P-Kt currently focuses on supporting knowledge representation and automatic reasoning via logic programming. The modular, unopinionated architecture of 2P-Kt is deliberately aimed at supporting and encouraging extensions towards other sorts of symbolic AI systems than Prolog—including PLP, which is currently missing. Accordingly in the following, we discuss how a module for ProbLog can actually be designed and realised to enrich the 2P-Kt ecosystem.

3. Design of Probabilistic Solver Module

Here we discuss how the 2P-Kt ecosystem can be enriched to support PLP. In particular, our goal is to add two major facilities to the ecosystem, namely: *(i)* a general-purpose API for probabilistic resolution, and *(ii)* a purpose-specific API for ProbLog-like resolution. Of course, while pursuing this purpose, the underlying technical requirement is to re-use the pre-existing facilities offered by 2P-Kt as much as possible. This includes terms, clauses, and theories representation, as well as Prolog's SLDNF resolution.

Accordingly, as depicted in Figure 1, PLP support is injected into the ecosystem via multiple self-contained and inter-dependent modules, each one representing a contribution of our proposal. Arrows indicate direct dependencies from one module to another. Of course, dependencies are *transitive*, meaning that each module inherits (and can therefore exploit) all the facilities carried by the other modules it depends upon, either directly or indirectly. Notably, PLP related modules are: `:bdd`, `:solve-plp`, `:solve-problog`, and `:ide-plp`.

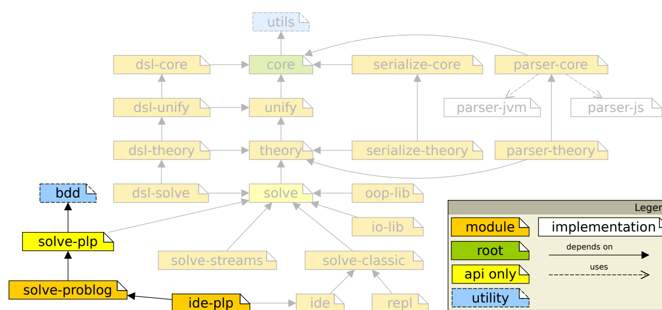


Figure 1: Architectural overview of our PLP and ProbLog modules, and their role within the 2P-Kt ecosystem

The `:bdd` module represents our proposal for the binary decision diagram manipulation library. This module is purely self-contained, in the sense that it does not rely upon any external

[1]https://kotlinlang.org/docs/mpp-supported-platforms.html

facility to support BDD. Rather, it consists of a pure Kotlin solution, which therefore puts no additional constraint on the platforms targetted by 2P-Kt. It is worth noting that, with such a choice, we intend to promote the usage of the library as a lean external dependency on other projects as well.

The `:solve-plp` module is meant to bundle all the entities and traits that are common to any potential implementation of solvers for the PLP paradigm. In other words, it is where our goal *(i)* is realised. This is a purely abstract module, that only provides API, interfaces and classes on which multiple PLP solver implementations can rely upon. Notable, this module depends on 2P-Kt's `:solve` module, which provides common abstractions for logic solvers and fixes their API, in order to keep them interoperable. In other words, we model *probabilistic* logic solvers as a direct subset of logic solvers.

The `:solve-problog` module contains the actual implementation of the PLP solver supporting the ProbLog language. In other words, this is where our goal *(ii)* is realised. As ProbLog solvers will be particular cases of probabilistic solvers, the `:solve-problog` module depends on the abstractions of `:solve-plp` and it is compliant to them. The other fundamental (and indirect) dependency is the `:bdd` module, which is used for manipulating binary decision diagrams during probabilistic logic goal resolution. Additionally, it also depends on `:solve-classic`— as ProbLog solvers will exploit ordinary Prolog resolution behind the scenes. Further details about the inner design and functioning of this module are discussed in the remainder of this section, and represent the main contribution of this paper.

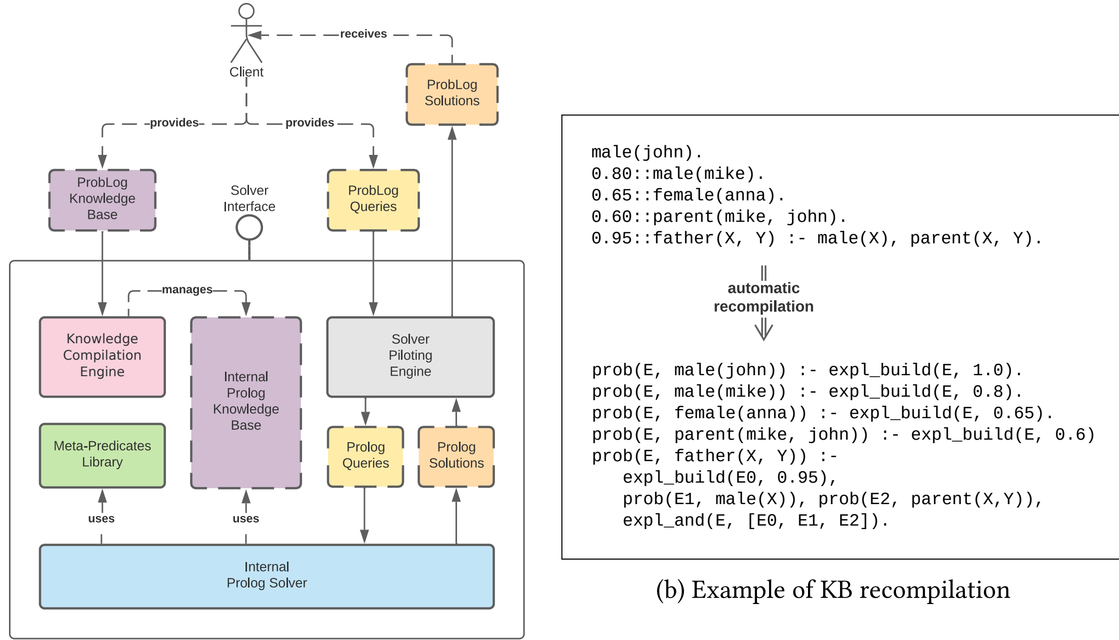
Finally, the `:ide-plp` module implements a stand-alone graphical application based on JavaFX, aimed at letting 2P-Kt users practice with ProbLog via an integrated environment.

3.1. Design Rationale

Figure 3a provides an overview of the overall design of our `:solve-problog` module. Overall, the module aims at providing a notion of ProbLog solver as a particular case of logic solvers. As any other sort of solver in 2P-Kt, ProbLog solvers accept users' queries as inputs – consisting of (possibly partially instantiated) logic atoms – and produce a multitude of solutions as outputs— consisting of variable assignments and probabilities. Notably, solutions are computed against a ProbLog knowledge base, which, in practice, consists of a Prolog theory with annotated clauses.

To perform *probabilistic* resolution, each ProbLog solver relies on a Prolog solver behind the scenes. The Prolog solver expects the probabilistic theory to be compiled into an ordinary Prolog theory aimed at constructing a BDD as the resolution process proceeds. In this phase, each probabilistic clause of the form p`::`*`Head`*` :- `*`Body`* is transformed into and ordinary Prolog clause of the form `prob(`*`Explanation`*`, `*`Head`*`) :- `*`Body2`*, where *`Explanation`* represents the BDD to be constructed out of the probability p and *`Body`*, whenever the probability of some sub-goal *`Head`* must be computed. A number of *ad-hoc* meta-predicates can be exploited in the clauses' bodies to serve the purpose of incrementally building a BBD. Under such assumption, the underlying Prolog solver may answer to probabilistic queries of the form `prolog_query(-`*`Probability`*`, +`*`Goal`*`)`. More precisely, the `prolog_query/2` predicate is in charge of *(i)* computing all possible Prolog solutions for *`Goal`* and *(ii)* constructing their specific BDD, then *(iii)* merging them into a unique BDD aimed at computing the overall *`Probability`* of *`Goal`*.

Figure 2: Architecture of our ProbLog solver (left), with a focus on the KB recompilation step (right).



(a) Architecture and information flow of our ProbLog solver

(b) Example of KB recompilation

To sum up, a ProbLog solver is a bi-directional façade among the user and the underlying Prolog solver. It takes care of translating probabilistic theories and queries in Prolog form, and Prolog solutions back into probabilistic form. Given this overview, the design of our PLP solver is built on top of three interconnected components: *(i)* a knowledge compilation engine, *(ii)* a library of meta-predicates, and *(iii)* a solver piloting engine. In the remainder of this section, we delve into the details of these components.

3.1.1. Knowledge Compilation Engine.

Each ProbLog solver of ours is backed by a Prolog solver aimed at computing an *explanation* (i.e. a BDD) for each possible probabilistic query. However, the Prolog solver can only deal with ordinary logic theories consisting of unannotated Horn clauses. Accordingly, *knowledge compilation engine* is the architectural component in charge of converting annotated probabilistic theories provided by the ProbLog users into ordinary Prolog users. It does so by applying a number of rewriting rules to the probabilistic theory:

$$
\begin{aligned}
[\![f(\bar{X}).]\!] &\longrightarrow \text{`}\texttt{prob(}E\texttt{, } f(\bar{X})\texttt{) :- expl_build(}E\texttt{, 1.0).}\text{'} \\
[\![p\texttt{::}f(\bar{X}).]\!] &\longrightarrow \text{`}\texttt{prob(}E\texttt{, } f(\bar{X})\texttt{) :- expl_build(}E\texttt{, } p\texttt{).}\text{'} \\
[\![p\texttt{::}f(\bar{X}) \texttt{ :- } b_1(\bar{X}_1)\texttt{, ..., } b_n(\bar{X}_n).]\!] &\longrightarrow \text{`}\texttt{prob(}E\texttt{, } f(\bar{X})\texttt{) :- expl_build(}E_0\texttt{, } p\texttt{),} \\
&\qquad \texttt{prob(}E_1\texttt{, } b_1(\bar{X}_1)\texttt{), ..., prob(}E_n\texttt{, } b_n(\bar{X}_n)\texttt{),} \\
&\qquad \texttt{expl_and(}E\texttt{, [}E_0\texttt{, } E_1\texttt{, ..., } E_n\texttt{]).}\text{'} \\
[\![p_1\texttt{::}f_1(\bar{X}_1)\texttt{, ..., } p_m\texttt{::}f_m(\bar{X}_m) \texttt{ :- } \bar{b}.]\!] &\longrightarrow \text{`}[\![p_1\texttt{::}f_1(\bar{X}_1) \texttt{ :- } \bar{b}.]\!]. \ldots [\![p_m\texttt{::}f_m(\bar{X}_m) \texttt{ :- } \bar{b}.]\!].\text{'}
\end{aligned}
$$

There, the first rule handles the case of unannotated facts (a.k.a. evidence). They are considered as certain facts—i.e. facts having 1.0 as probability. The second rule handles the case of annotated facts having a probability $p \in [0, 1] \subset \mathbb{R}$. Finally, the third rule handles the case of annotated rules, whereas the last rule handles the case of probabilistic clauses having annotated disjunctions in their heads. Because of space limitations, we here omit other rules aimed at handling conjunction, negation, or implication in clauses' bodies. In all such cases, $f, f_1, \ldots, f_m, b_1, \ldots, b_n$ denote logic predicates' symbols of arbitrary arity, $p, p_1, \ldots, p_m$ are real numbers in the $[0, 1]$ range denoting probability values, $\bar{X}, \bar{X}_1, \ldots, \bar{X}_n, \bar{X}_m$ denote tuples of logic terms of arbitrary length, while $\bar{b}$ denote a conjunction of logic atoms involving zero, one, or more atoms.

Figure 3b exemplifies the knowledge compilation engine in action on a simple probabilistic theory. As the reader may notice, the resulting Prolog theory consists of a number of rules of the form `prob(-Explanation, +Goal)`, aimed at computing the an *`Explanation`* for a particular *`Goal`*. The bodies of such rules may exploit a number of built-in meta-predicates aimed at iteratively constructing an explanation out of simpler explanations.

3.1.2. Library of Meta-Predicates

A fundamental prerequisite for the knowledge compilation engine to work is that BDD can be suitably represented in logic, as explanations, at the end of the day, consist of BDD instances.

To address such a need, in our `:solve-plp` module, we define a whole new class of logic constants aimed at *referencing* particular instances of BDD. So BDD instances – which are in-memory data structures in Kotlin's object-oriented world – are treated as constants in the logic realm. In this way, BDD instances can be carried around, constructed, or composed as part of resolution, and possibly bound to variables such as *`Explanation`* or $E, E_1, \ldots, E_n$ mentioned above.

To make it possible to create and compose BDD from a logic program, we introduced a library of Prolog-compliant meta-predicates, each one implementing a specific task supporting ProbLog-like inference. Of course, as such meta-predicates operate on data structures that lay outside the logic realm, they cannot be defined in Prolog. Accordingly, through the *generator* mechanism of 2P-Kt (cf. [30]), we are able to implement the behaviour of these meta-predicates with object-oriented Kotlin code. At the functional level, however, the behaviour of most relevant meta-predicates can be described as follows:

expl_and(-E, [+E_1, ..., +E_n]) | provided that variables $E_1, \ldots, E_n$ are bound to as many constants representing $n \geq 2$ BDD, this meta-predicate merges them all into a new BDD representing their conjunction, and binds a constant to E referencing that BDD

expl_or(-E, [+E_1, ..., +E_n]) | like the above, but for disjunction

expl_not(-E, +E') | like the above, but for negation

expl_build(-E, +P) | provided that variable E is bound to a number representing a valid probability value, this meta-predicate creates a bare new, minimal BDD out of that probability value, and binds a constant to E referencing that BDD

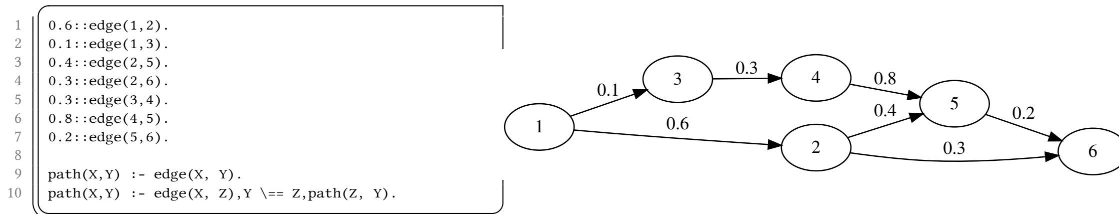


Figure 4: Example of Probabilistic Graph Modeling: ProbLog syntax (left) and corresponding graph (right)

The `prolog_query(-Probability, +Goal)` meta-predicate then closes the loop, acting as the main entry point for probabilistic resolution in Prolog. The first argument represents the numeric probability of the goal being queries, and the second argument is the goal itself. The probability argument can either be an input number or an output variable. If the goal argument is a non-ground term, its variables are substituted for each solution found by the solver. Of course, despite the `prolog_query/2` meta-predicate simulates a lazy enumeration of solutions via backtracking, the whole set of solutions must be eagerly computed behind the scenes, in order to compute probabilities. Hence, queries having an infinite proof tree may lead to a situation where the solver gets stuck or saturates the available memory even before the first solution is presented to the user.

3.1.3. Solver Piloting Engine

The last piece needed by our system to fully implement a PLP inference solver is a component aimed at hiding the presence of an underlying Prolog solver. We call this component the *solver piloting engine*.

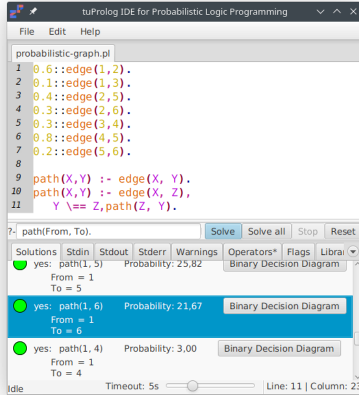
This component is responsible for accepting LP and PLP queries from clients, properly configuring the underlying LP solver, piloting it to infer the solutions, extracting the probability values and presenting the results. As a matter of fact, it represents the presentation layer of our system. Notably, solver configurations are handled at this level, and the component is capable of passing both LP and PLP queries to the inner solver on demand.

Also, the solver piloting engine recompiles queries and goals bidirectionally to be compliant with the meta-predicates semantics of our solution. For instance, our solution assumes that each query is represented via the `prob_query/2` predicate. Considering the example in Figure 4, a query such as `path(From, To)` would be transformed in `prob_query(P, path(From, To))`. Once solutions are found, the *solver piloting engine* extracts the two terms *P* and `path(From, To)`, and presents their values to the clients in the correct format.

4. Multi-platform Support Demonstration

Here we provide a demonstration of our ProbLog module for 2P-Kt. More precisely, we show how our solution supports: *(i)* a wide gamma of usage modalities – ranging from issuing ProbLog

(a)



(b)

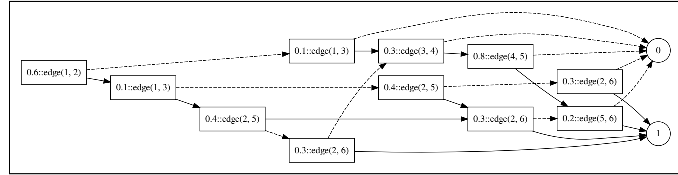


Figure 5: 2P-Kт PLP IDE (5a) and corresponding BDD built by the solver (5b)

queries via a GUI to usage "as a library" –, and *(ii)* a number of mainstream programming platforms and languages.

In particular, our demonstration works by solving a probabilistic query against the trivial probabilistic logic program from Figure 4 – where a probabilistic graph is modelled in ProbLog –, enumerating all possible solutions and interpreting them as possible paths and their probabilities. We perform this action multiple times, and in several ways, each time showing a different usage modality. Notably, we exemplify the usage ProbLog as a JavaFX-based graphical application, other than as a Kotlin, Java, Android, Python, and JavaScript library.

Figure 5 shows 2P-Kт's IDE, tailored on our ProbLog module. The whole demonstration can be reproduced by downloading the PLP IDE executable (`2p-ide-plp-X.Y.Z-redist.jar`) from https://github.com/tuProlog/2p-kt/releases/latest. The IDE accepts ProbLog theories as input, either from a file or as bare textual input, and it is designed to resemble a simple text editor. One can issue a query and submit it to the underlying ProbLog solver. Once computed, solutions to that query are shown in a list view. Also, a tab view enables the inspection of the internal state of the solver. Figure 5b depicts the BDD used by our ProbLog solver behind the scenes while computing the probability of solution `path(1, 6)`. Notably, BDD representation is yet another function of our IDE, attained via an automatically-generated DOT [31] specification.

Figure 6 shows how our ProbLog module can be used "as a library" on multiple programming platforms and languages, namely Kotlin (for both the JVM and Android platforms), Python, and JavaScript. The Java language is supported as well, despite not being depicted in the figure. The similarity among the code snippets is deliberate and aimed at stressing how the many 2P-Kт ports share a common design and API, despite the slight syntactical differences characterising the target language. The conceptual flow is analogous: *(i)* a `ClausesParser` is instantiated out of the set of ProbLog predicates (i.e. Prolog's standard predicates, plus `::/2`), *(ii)* it is then used to parse the ProbLog program from Figure 4, *(iii)* the resulting `Theory` is used as *static* KB of a newly instantiated ProbLog `Solver`, *(iv)* the query `path(`*`From`*`,`*`To`*`)` is

```
// Kotlin
val clausesParser = ClausesParser.withOperators(PROBLOG_OPERATORS)
val probabilisticTheory = clausesParser.parseTheory("⟨theory from fig. 4⟩")
val problogSolver = Solver.problog.solverWithDefaultBuiltins(staticKb = probabilisticTheory)
val goal = Struct.of("path", Var.of("From"), Var.of("To"))
for (solution in problogSolver.solve(goal, SolveOptions.allLazily().probabilistic()))
    if (solution.isYes)
        println("yes: ${solution.solvedQuery} with probability ${solution.probability}")
```

```
# Python
probabilisticTheory = parse_theory("⟨theory from fig. 4⟩", PROBLOG_OPERATORS)
problogSolver = problog_solver(static_kb=probabilisticTheory)
query = struct('path', var('From'), var('To'))
for solution in problogSolver.solve(query, solve_options(lazy=True, probabilistic=True)):
    if solution.is_yes:
        print(f"yes: {solution.solved_query} with probability {probability(solution)}")
```

```
// JavaScript
let clausesParser = ClausesParser.Companion.withOperatorSet(PROBLOG_OPERATORS)
let scope = Scope.Companion.empty()
let probabilisticTheory = clausesParser.parseTheory("⟨theory from fig. 4⟩")
let problogSolver = Solver.Companion.problog.solverWithDefaultBuiltinsAndStaticKB(probabilisticTheory)
let query = scope.structOf("path", [scope.varOf("From"), scope.varOf("To")])
let options = probabilistic(SolveOptions.Companion.allLazily())
let si = problogSolver.solveWithOptions(query, options).iterator()
while (si.hasNext()) {
    let solution = si.next();
    if (solution.isYes)
        console.log('yes: ${solution.solvedQuery} with probability ${probability(solution)}')}
```

Figure 6: Usage of 2P-Kt's ProbLog module "as a library"

programmatically constructed, and *(v)* issued to the `Solver`, as a *probabilistic* query. Solutions are then *(vi)* enumerated, and, finally, *(vii)* positive solutions are printed, along with their *probabilities*. For the sake of reproducibility, the provided snippets can be executed on all the supported platforms by cloning the Git repository https://github.com/tuProlog/2pkt-problog-compatibility-demo, and by following the contained instruction. As the reader may easily observe, the resulting solutions and probabilities are the same depicted in Figure 5a.

5. Conclusion and Future Works

This paper describes the design and implementation of a ProbLog solver as a module of a logic ecosystem. The extension pursues the twofold goal of *(i)* enriching the 2P-Kt logic ecosystem and technology towards PLP and, in particular, ProbLog, and *(ii)* bridging PLP and main-stream programming platforms and languages by letting developers benefit from a *library* providing probabilistic reasoning capabilities to their projects.

The proposed solution is still in its infancy, and it is still not suitable to be compared with other proposals in the field—at least for what concerns performance or feature richness. However, by working on top of the 2P-Kt ecosystem, our solution inherits large platform support – as demonstrated in this paper –, thus overcoming the usability and portability constraints that affect other solutions in this field. In fact, our technology can be deployed on all the platforms supported by 2P-Kt—which currently include, but are not limited to, the JVM, Android, Python, and JavaScript. In the long term, we believe such technological openness will play a fundamental role in bringing the benefits of (P)LP to the general public and letting AI practitioners exploit

(P)LP with minimal effort. In this perspective, our proposal represents a first step in this direction.

Ultimately, one of the top priorities of this research effort is to leave the door open to future developments. Our design is purposely abstract, and we endorse the future exploration of alternative implementation ideas. Among the others, we envision future directions involving: approximate inference support, more efficient knowledge compilation data structures, or the exploitation alternative resolution strategies such as the tabled or concurrent ones—other than, of course, comparative benchmarks aimed at assessing our solutions w.r.t. the state of the art.

Acknowledgments

The project has been supported by the H2020 ERC Project "CompuLaw" (G.A. 833647).

References

[1] R. Calegari, G. Ciatto, V. Mascardi, A. Omicini, Logic-based technologies for multi-agent systems: A systematic literature review, Autonomous Agents and Multi-Agent Systems 35 (2021) 1:1–1:67. doi:10.1007/s10458-020-09478-3, collection "Current Trends in Research on Software Agents and Agent-Based Software Development".

[2] R. Calegari, A. Omicini, G. Sartor, Explainable and ethical AI: A perspective on argumentation and logic programming, in: M. Baldoni, S. Bandini (Eds.), AIxIA 2020 – Advances in Artificial Intelligence, volume 12414 of *Lecture Notes in Computer Science*, Springer Nature, 2021, pp. 19–36. doi:10.1007/978-3-030-73065-9_2.

[3] A. Ortega, J. Fierrez, A. Morales, Z. Wang, T. Ribeiro, Symbolic ai for xai: Evaluating lfit inductive programming for fair and explainable automatic recruitment, in: IEEE/CVF Winter Conference on Applications of Computer Vision (WACV), IEEE, 2021, pp. 78–87. doi:10.1109/WACVW52041.2021.00013.

[4] R. Calegari, G. Ciatto, E. Denti, A. Omicini, Logic-based technologies for intelligent systems: State of the art and perspectives, Information 11 (2020) 1–29. doi:10.3390/info11030167, special Issue "10th Anniversary of Information—Emerging Research Challenges".

[5] R. Calegari, G. Ciatto, A. Omicini, On the integration of symbolic and sub-symbolic techniques for XAI: A survey, Intelligenza Artificiale 14 (2020) 7–32. doi:10.3233/IA-190036.

[6] C. Perez, The deep learning AI playbook, Lulu.com, 2017.

[7] R. Ng, V. S. Subrahmanian, Probabilistic logic programming, Information and computation 101 (1992) 150–201. doi:10.1016/0890-5401(92)90061-J.

[8] F. Riguzzi, Foundations of Probabilistic Logic Programming, River Publishers, Gistrup, Denmark, 2018. URL: http://www.riverpublishers.com/book_details.php?book_id=660.

[9] L. De Raedt, A. Kimmig, H. Toivonen, Problog: A probabilistic prolog and its application in link discovery, in: M. M. Veloso (Ed.), IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007, pp. 2462–2467. URL: http://ijcai.org/Proceedings/07/Papers/396.pdf.

[10] F. Riguzzi, A top down interpreter for LPAD and cp-logic, in: R. Basili, M. T. Pazienza (Eds.), AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, 10th Congress of the Italian Association for Artificial Intelligence, Rome, Italy, September 10-13, 2007, Proceedings, volume 4733 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 109–120. doi:10.1007/978-3-540-74782-6_11.

[11] A. Kimmig, B. Demoen, L. De Raedt, V. S. Costa, R. Rocha, On the implementation of the probabilistic logic programming language problog, Theory and Practice of Logic Programming 11 (2011) 235–262. doi:10.1017/S1471068410000566.

[12] F. Niu, C. Zhang, C. Ré, J. Shavlik, Elementary: Large-scale knowledge-base construction via machine learning and statistical inference, International Journal on Semantic Web and Information Systems (IJSWIS) 8 (2012) 42–73.

[13] G. Ciatto, R. Calegari, A. Omicini, 2P-Kt: A logic-based ecosystem for symbolic AI, SoftwareX 16 (2021) 100817:1–7. doi:10.1016/j.softx.2021.100817.

[14] D. Fierens, G. V. den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. D. Raedt, Inference and learning in probabilistic logic programs using weighted boolean formulas, Theory and Practice of Logic Programming 15 (2015) 358–401. doi:10.1017/S1471068414000076.

[15] F. Riguzzi, T. Swift, The PITA system for logical-probabilistic inference, in: S. H. Muggleton, H. Watanabe (Eds.), Latest Advances in Inductive Logic Programming, ILP 2011, Late Breaking Papers, Windsor Great Park, UK, July 31 - August 3, Imperial College Press / World Scientific, 2011, pp. 79–86. doi:10.1142/9781783265091_0010.

[16] T. Sato, A statistical learning method for logic programs with distribution semantics, in: L. Sterling (Ed.), Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, MIT Press, 1995, pp. 715–729.

[17] T. Sato, Y. Kameya, PRISM: A language for symbolic-statistical modeling, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 2 Volumes, Morgan Kaufmann, 1997, pp. 1330–1339. URL: http://ijcai.org/Proceedings/97-2/Papers/078.pdf.

[18] P. Horwich, Probability and evidence, Cambridge University Press, 2016. doi:10.1017/CBO9781316494219.

[19] L. De Raedt, A. Kimmig, Probabilistic (logic) programming concepts, Machine Learning 100 (2015) 5–47. doi:10.1007/s10994-015-5494-z.

[20] A. Darwiche, P. Marquis, A knowledge compilation map, Journal of Artificial Intelligence Research 17 (2002) 229–264. doi:10.1613/jair.989.

[21] S. B. Akers, Binary decision diagrams, IEEE Transactions on computers 27 (1978) 509–516. doi:10.1109/TC.1978.1675141.

[22] A. Lovato, D. Macedonio, F. Spoto, A thread-safe library for binary decision diagrams, in: D. Giannakopoulou, G. Salaün (Eds.), Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, volume 8702 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 35–49. doi:10.1007/978-3-319-10431-7_4.

[23] E. Bellodi, F. Riguzzi, Expectation maximization over binary decision diagrams for probabilistic logic programs, Intelligent Data Analysis 17 (2013) 343–363. doi:10.3233/IDA-130582.

[24] J. Vlasselaer, J. Renkens, G. Van den Broeck, L. De Raedt, Compiling probabilistic logic programs into sentential decision diagrams, in: Proceedings Workshop on Probabilistic Logic Programming (PLP), 2014, pp. 1–10.

[25] V. S. Costa, R. Rocha, L. Damas, The YAP prolog system, Theory Pract. Log. Program. 12 (2012) 5–34. doi:10.1017/S1471068411000512.

[26] J. Vennekens, M. Denecker, M. Bruynooghe, Cp-logic: A language of causal probabilistic events and its relation to logic programming, Theory Pract. Log. Program. 9 (2009) 245–308. doi:10.1017/S1471068409003767.

[27] A. Thayse, M. Davio, J. Deschamps, Optimization of multivalued decision algorithms, in: Proceedings of the eighth international symposium on Multiple-valued logic, MVL 1978, Rosemont, Illinois, USA, 1978, IEEE Computer Society Press, 1978, pp. 171–178. URL: http://dl.acm.org/citation.cfm?id=804202.

[28] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, Swi-prolog, Theory Pract. Log. Program. 12 (2012) 67–96. doi:10.1017/S1471068411000494.

[29] G. Ciatto, R. Calegari, E. Siboni, E. Denti, A. Omicini, 2P-Kt: logic programming with objects & functions in Kotlin, in: R. Calegari, G. Ciatto, E. Denti, A. Omicini, G. Sartor (Eds.), WOA 2020 – 21th Workshop "From Objects to Agents", volume 2706 of *CEUR Workshop Proceedings*, Sun SITE Central Europe, RWTH Aachen University, Aachen, Germany, 2020, pp. 219–236. URL: http://ceur-ws.org/Vol-2706/paper14.pdf, 21st Workshop "From Objects to Agents" (WOA 2020), Bologna, Italy, 14–16 September 2020. Proceedings.

[30] G. Ciatto, R. Calegari, A. Omicini, Lazy stream manipulation in Prolog via backtracking: The case of 2p-kt, in: W. Faber, G. Friedrich, M. Gebser, M. Morak (Eds.), Logics in Artificial Intelligence, volume 12678 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 407–420. doi:10.1007/978-3-030-75775-5_27, 17th European Conference, JELIA 2021, Virtual Event, May 17–20, 2021, Proceedings.

[31] E. R. Gansner, S. C. North, An open graph visualization system and its applications to software engineering, SoftwareE - Practice and Experience 30 (2000) 1203–1233. doi:10.1002/1097-024X(200009)30:11\%3C1203::AID-SPE338\%3E3.0.CO;2-N.