ProbLog Documentation

Release 2.1

KU Leuven, DTAI Research Group

Jul 23, 2021

Contents

Welcome to the ProbLog 2.1 documentation.

Contents:

CHAPTER 1

Installing ProbLog

1.1 Prerequisites

ProbLog is built on Python. ProbLog is compatible with Python 3.

Python is included in most installations of Linux and Mac OSX. Windows users can find instructions on how to install it in the Python documentation.

1.2 Installing with pip

ProbLog is available in the Python Package Index (PyPi) and it can be installed with

```
pip install problog
```

To install as user without root permissions.

```
pip install problog --user
```

After installation as user you may need to add the location of the `problog` script to your PATH. Common location for this script are `~/.local/bin` or `~/Library/Python/2.7/bin/`.

To update ProbLog to the latest version.

```
pip install problog --upgrade
```

To install the latest ProbLog development version.

```
pip install problog --pre --upgrade
```

To install ProbLog with support for Sentential Decision Diagrams (currently not supported on Windows).

```
pip install problog[sdd]
```

CHAPTER 2

ProbLog models

2.1 Prolog

The ProbLog modeling language is based on Prolog.

For a very quick introduction to Prolog you can check the Wikipedia page.

For a more in-depth introduction you can check the Learn Prolog Now! tutorial or the book Simply Logical by Peter Flach.

2.2 ProbLog

ProbLog extends Prolog syntax by introducing few new operators to allow for probabilistic modeling. The following table provides a simple overview.

Definition	Example
fact	`a.`
probabilistic fact	`0.5::a.`
clause	`a :- x.`
probabilistic clause	`0.5::a :- x.`
annotated disjunction	`0.5::a; 0.5::b.`
annotated disjunction	`0.5::a; 0.5::b :- x.`

In addition, Problog introduces also two predicates `query` and `evidence` for querying a probabilistic program or to condition it to some pieces of evidence.

2.2.1 Probabilistic facts

The main difference between Prolog and ProbLog is that ProbLog support probabilistic facts. In the language, this extension is realized by the addition of a single operator `::`.

In an example program involving coin tosses, we could have the following statement.

```
0.5::heads.
```

This indicates that the fact *heads* is true with probability 0.5 and false with probability 1-0.5.

This statement introduces *one* probabilistic choice. If we want to model two coins, we need two separate facts:

```
0.5::heads1.
0.5::heads2.
```

We can generalize this to an unbound number of coins by using a variable argument:

```
0.5::heads(C).
```

2.2.2 Annotated disjunctions

ProbLog also supports non-binary choices. For example, we can model the throw of die as follows.

```
1/6::die(D, 1); 1/6::die(D, 2); 1/6::die(D, 3);
1/6::die(D, 4); 1/6::die(D, 5); 1/6::die(D, 6).
```

This type of statement is called an *annotated disjunction*. It expresses that at most one of these choices is true. There is always an implicit *null* choice which states that none of the options is taken. In this example, however, that extra state has zero probability because the probabilities of the other states sum to one.

2.2.3 Probabilistic clauses

ProbLog also supports probabilities in the head of clauses.

```
0.1::burglary.
0.9::alarm :- burglary.
```

This means that if burglary is true, alarm will be true as well with 90% probability. Such a program can always be transformed into a program with just probabilistic facts.

```
0.1::burglary.
0.9::alarm_on_burglary.

alarm :- burglary, alarm_on_burglary.
```

Similarly, annotated disjunctions can also be used as head of a clause.

```
0.5::weather(0,sun); 0.5::weather(0,rain).
0.8::weather(T,sun); 0.2::weather(T,rain) :- T > 0, T1 is T - 1, weather(T1, sun).
0.4::weather(T,sun); 0.6::weather(T,rain) :- T > 0, T1 is T - 1, weather(T1, rain).
```

This program can also be transformed into an equivalent program with only annotated disjunctive facts.

```
0.5::weather(0,sun); 0.5::weather(0,rain).

0.8::weather_after_sun(T,sun); 0.2::weather_after_sun(T,rain).
weather(T, sun) :- T > 0, T1 is T - 1, weather(T1, sun), weather_after_sun(T, sun).
weather(T, rain) :- T > 0, T1 is T - 1, weather(T1, sun), weather_after_sun(T, rain).
```

(continues on next page)

(continued from previous page)

```
0.4::weather_after_rain(T,sun); 0.6::weather_after_rain(T,rain).
weather(T, sun) :- T > 0, T1 is T - 1, weather(T1, sun), weather_after_rain(T, sun).
weather(T, rain) :- T > 0, T1 is T - 1, weather(T1, sun), weather_after_rain(T, rain).
```

2.2.4 Queries

A query indicates for which entity we want to compute the probability.

Queries are specified by adding a fact `query(Query)`:

```
0.5::heads(C).
two_heads :- heads(c1), heads(c2).
query(two_heads).
```

Queries can also be added in batch.

```
0.5::heads(C).
query(heads(C)) :- between(1, 4, C).
```

This will add the queries `heads(1)`, `heads(2)`, `heads(3)` and `heads(4)`.

It is also possible to give a non-ground query, on the condition that the program itself contains sufficient information to ground the probabilistic parts.

```
0.5::heads(C) :- between(1, 4, C).
query(heads(C)).
```

This has the same effect as the previous program.

2.2.5 Evidence

Evidence specifies any observations on which we want to condition this probability. Evidence conditions a part of the program to be true or false.

It can be specified using a fact `evidence(Literal)`.

```
0.5::heads(C).
two_heads :- heads(c1), heads(c2).
evidence(\+ two_heads).
query(heads(c1)).
```

This program computes the probability that the first coin toss produces heads when we know that the coin tosses did not both produce heads. You can try it out in the online editor.

Evidence can also be specified using the binary predicate `evidence(Positive, true)` and `evidence(Positive, false)`.

2.2.6 Tabling

In ProbLog everything is tabled (or memoized). Tabling is an advanced form of caching that is used to speed-up the execution of logic programs and that allows certain types of cyclic programs.

Consider for example the following program that computes Fibonacci numbers.

```
fib(1, 1).
fib(2, 1).
fib(N, F) :-
    N > 2,
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2.
```

In standard Prolog the execution time of this program is exponential in the size of N because computations are not reused between recursive calls. In tabled Prolog, the results of each computation is stored and reused when possible. In this way, the above program becomes linear.

The previous example shows the power of caching, but tabling goes further than that. Consider the following program that defines the ancestor relation in a family tree.

```
parent(ann, bob).
parent(ann, chris).
parent(bob, derek).

ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).
ancestor(X, Y) :- parent(X, Y).
```

We want to find out the descendents of Ann (i.e. the query *ancestor(ann, X)*). In standard Prolog this program goes into an infinite recursion because the call to *ancestor(ann, X)* leads immediately back to the equivalent call *ancestor(ann, Z)*.

In tabled Prolog, the identical call is detected and postponed, and the correct results are produced.

Another example is that of finding a path in a (possibly cyclic) graph. In ProbLog (or any other tabled Prolog) you can simply write.

```
path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
```

2.2.7 Control predicates

ProbLog uses Prolog to generate a ground version of a probabilistic logic program. However, it does not support certain features that have no meaning in a probabilistic setting. This includes cuts (`!`) and any other mechanism that breaks the pure logic interpretation of the program.

For a full list of features that ProbLog does (not) support, please check *this section*.

2.2.8 Findall

ProbLog supports the meta-predicate `findall/3` for collecting all results to a query. It is similar to `findall/3` in Prolog, but it eliminates duplicate solutions (so it corresponds to `all/3` in YAP Prolog).

Note that the use of findall can lead to a combinatorial explosion when used in a probabilistic context.

2.3 Other modes syntax

When ProbLog is executed in modes that are different from standard inference, new specific notation is available.

2.3.1 Learning from interpretations (LFI) mode

ProbLog programs can be used in a learning setting, where some or all the probabilities are unkonwn. In this case, the probability annotation in a probabilistic fact can be one of three possible forms:

- Of the form `t(_)`, as in for instance `t(_)::p_alarm1`. This indicates that the probability of this fact is to be learned from data.
- Of the form `t(p)`, with `p` a probability, as in for instance `t(0.5)::burglary`. This again indicates that the probability of this fact is to be learned from data, but instead of initializing this probability randomly, it will be set to the value `p` in the first iteration of EM.
- Of the form `p`, with `p` a probability, as in for instance `0.2::earthquake`. This indicates that the probability of this fact is fixed (not learned), and it correspond to the standard annotation of probabilistic facts.

In a learning setting, the ProbLog model is usually accompanied with a set of examples to learn from. Examples are provided using the `evidence` predicate for each atom in an example. Examples are separated using dashes `---`.

An example of learning model:

```
t(_)::heads1.
t(_)::heads2.
someHeads :- heads1.
someHeads :- heads2.
```

An example of how to provide examples:

```
evidence(someHeads,false).
evidence(heads1,false).
----------------
evidence(someHeads,true).
evidence(heads1,true).
----------------
evidence(someHeads,true).
evidence(heads1,false).
----------------
```

2.3.2 Decision-theoretic mode

DTProbLog is a decision-theoretic extension of ProbLog.

A model in DTProbLog differs from standard ProbLog models in a number of ways:

- There are no queries and evidence.
- Certain facts are annotated as being a decision fact for which the optimal choice must be determined.
- Certain atoms are annotated with an utility, indicating their contribution to the final score.

Decision facts can be annotated in any of the following ways:

```
?::a.
decision(a).
```

Utilities can be defined using the `utility/2` predicate:

```
utility(win, 10).
utility(buy, -1).
```

2.4 Libraries and Builtins

ProbLog has a number of builtins and libraries available that simplify modeling. An overview can be found on the page *Builtins and Libraries*.

CHAPTER 3

Using ProbLog as a standalone tool

The command line interface (CLI) gives access to the basic functionality of ProbLog 2.1. It is accessible through the script `problog` (or `problog-cli.py` in the repository version).

The CLI has different modes of operation. These can be accessed by adding a keyword as the first argument.

Currently, the following modes are supported

- (default, no keyword): standard ProbLog inference
- `sample`: generate samples from a ProbLog program
- `mpe`: most probable explanation
- `lfi`: learning from interpretations
- `dt`: decision-theoretic problog
- `map`: MAP inference
- `explain`: evaluate using mutually exclusive proofs
- `ground`: generate a ground program
- `bn`: export a Bayesian network
- `shell`: interactive shell
- `install`: run the installer
- `unittest`: run the testsuite
- `web`: start a web server

3.1 Default (no keyword)

Run ProbLog in standard inference mode.

Used as `problog <model> [optional]` where:

- <model> is a file containing the ProbLog model;
- [optional] is a set of optional parameters.

Returns a set of probabilities for the queries in the model.

For example, given a file some_heads.pl

```
$ cat some_heads.pl
0.5::heads1.
0.6::heads2.
someHeads :- heads1.
someHeads :- heads2.

query(someHeads).
```

We can do

```
$ problog some_heads.pl
someHeads : 0.8
```

This mode supports many optional arguments to customize the inference:

- --knowledge {sdd,sddx,bdd,nnf,ddnnf,kbest,fsdd,fbdd}, -k {sdd,sddx,bdd, nnf,ddnnf,kbest,fsdd,fbdd}; Knowledge compilation tool. By default, it uses the first available option from SDD, d-DNNF using c2d and d-DNNF using dsharp.
- --combine Combine input files into single model.
- --logspace Use log space evaluation (default).
- --nologspace Use normal space evaluation.
- --symbolic Use symbolic computations.
- --output OUTPUT, -o OUTPUT; Output file (default stdout)
- --recursion-limit RECURSION_LIMIT; Set Python recursion limit. (default: 10000)
- --timeout TIMEOUT, -t TIMEOUT; Set timeout (in seconds, default=off).
- --compile-timeout COMPILE_TIMEOUT; Set timeout for compilation (in seconds, default=off).
- --debug, -d Enable debug mode (print full errors).
- --full-trace, -T Full tracing.
- -a ARGS, --arg ARGS Pass additional arguments to the cmd_args builtin.
- --profile output runtime profile
- --trace output runtime trace
- --profile-level PROFILE_LEVEL
- --format {text,prolog}
- -L LIBRARY, --library LIBRARY; Add to ProbLog library search path
- --propagate-evidence; Enable evidence propagation
- --dont-propagate-evidence; Disable evidence propagation
- --propagate-weights; Enable weight propagation
- --convergence CONVERGENCE, -c CONVERGENCE; Stop anytime when bounds are within this range

3.2 Sampling (`sample`)

Run ProbLog in sampling mode, generating possible assignments to the queries in the model.

Used as `problog sample <model> [optional]` where:

- `<model>` is a file containing the ProbLog model with the queries of interest;
- `[optional]` is a set of optional parameters.

For example, given a file `some_heads.pl`

```
$ cat some_heads.pl
0.5::heads1.
0.6::heads2.
someHeads :- heads1.
someHeads :- heads2.

query(someHeads).
```

We can do:

```
$ problog sample some_heads.pl -N 3
====================
% Probability: 0.2
====================
someHeads.
% Probability: 0.2
====================
someHeads.
% Probability: 0.3
```

The probability indicated is the probability of *the choices made to obtain the sample*. It is **NOT** the probability of the sample itself (because there may be multiple choices that lead to the same sample).

By default, only query atoms are part of the sample. To also include facts that were chosen while sampling, the argument `--with-facts` can be used. The result above would then become

```
$ problog sample some_heads.pl -N 3 --oneline --with-facts
% Probability: 0.2
heads1. someHeads. % Probability: 0.2
heads2. someHeads. % Probability: 0.3
```

The sampling algorithm supports **evidence** through rejection sampling. All generated samples are guaranteed to satisfy the evidence. Note that this process can be slow if the evidence has low probability.

The sampling algorithm support evidence propagation, that is, in certain cases it can ensure the evidence holds without the use of rejection sampling. To enable this feature use the `--propagate-evidence` argument. Evidence propagation is not supported on programs with continuous distributions, or on programs where the evidence has infinite support.

All the optional arguments:

- `-h, --help`; show the help message and exit
- `-N N, -n N`;Number of samples.
- `--with-facts`; Also output choice facts (default: just queries).
- `--with-probability`; Show probability.
- `--as-evidence`; Output as evidence.

- `--propagate-evidence`; Enable evidence propagation
- `--dont-propagate-evidence`; Disable evidence propagation
- `--oneline`; Format samples on one line.
- `--estimate`; Estimate probability of queries from samples (see next section).
- `--timeout TIMEOUT, -t TIMEOUT`; Set timeout (in seconds, default=off).
- `--output OUTPUT, -o OUTPUT`; Filename of output file.
- `--verbose, -v`; Verbose output
- `--seed SEED, -s SEED`; Random seed
- `--full-trace`;
- `--strip-tag`; Strip outermost tag from output.
- `-a ARGS, --arg ARGS`; Pass additional arguments to the cmd_args builtin.
- `--progress`; show progress.

3.2.1 Sample based inference

The sample mode can be used for *probability estimation* by setting the flag `--estimate`. The output is similar to the output in default mode.

The number of samples used for estimation can be determined in three ways:

- by supplying the number of samples using the argument `-N`
- by supplying a timeout using the argument `--timeout` or `-t` (not supported on Windows)
- by manually interrupting the process using CTRL-C or by sending a TERM(15) signal

```
$ problog sample some_heads.pl  --estimate -t 5
% Probability estimate after 7865 samples:
someHeads : 0.79249841
```

This mode also support the `--propagate-evidence` flag.

References:

Paper: https://lirias.kuleuven.be/handle/123456789/510199

Tutorial: https://dtai.cs.kuleuven.be/problog/tutorial/sampling/02_arithmeticexpressions.html

3.3 Most Probable Explanation (`mpe`)

Run ProbLog in MPE mode, computing the possible world with the highest probability in which all queries and evidence are true.

Used as `problog mpe <model> [optional]` where:

- `<model>` is a file containing the ProbLog model;
- `[optional]` is a set of optional parameters.

Returns:

- the possible world with the highest probability (as a set of facts);

- the probability of the most probable explanation.

The optional arguments are:

- `-h, --help`; show this help message and exit
- `--solver {maxsatz,scip,sat4j}`; MaxSAT solver to use
- `--full`; Also show false atoms.
- `-o OUTPUT, --output OUTPUT`; Write output to given file (default: write to stdout)
- `-v, --verbose`; Increase verbosity

For example, given a file `digraph.pl` describing a probabilistic graph:

```
$ cat digraph.pl
0.6::edge(1,2).
0.1::edge(1,3).
0.4::edge(2,5).
0.3::edge(2,6).
0.3::edge(3,4).
0.8::edge(4,5).
0.2::edge(5,6).

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), Y \== Z,path(Z,Y).

evidence(path(1,5)).
evidence(path(1,6)).
```

We can do:

```
$ problog mpe pgraph.pl
edge(4,5)  edge(1,2)  edge(2,5) edge(2,6)
\+edge(1,3)  \+edge(3,4)  \+edge(5,6)
% Probability: 0.0290304
```

3.4 Learning from interpretations (`lfi`)

Run ProbLog in the learning from interpretation (LFI) setting. Given a probabilistic program with parameterized weights and a set of (partial) interpretation, learns appropriate values of the parameters.

Used as: `problog lfi <model> <evidence> [optional]` where:

- `<model>` is the ProbLog model file;
- `<evidence>` is the a file containing a set of examples to learn from.
- `[optional]` are optional arguments

The command standard output is: `<loss> <probs> <atoms> <iter>` where:

- `<loss>` is the final loss of the learning problem;
- `<probs>` is a list of the learned paramenters (i.e. probabilities);
- `<atoms>` is the list of clauses that the probabilities refer to (positional mapping);
- `<iter>` is the number of EM iterations.

The optional arguments are:

- `-h, --help`; show the help message and exit
- `-n MAX_ITER`;
- `-d MIN_IMPROV`;
- `-O OUTPUT_MODEL, --output-model OUTPUT_MODEL`; write resulting model to given file
- `-o OUTPUT, --output OUTPUT`; write output to file
- `--logger LOGGER`; write log to a given file
- `-k {sdd,sddx,ddnnf}, --knowledge {sdd,sddx,ddnnf}`; knowledge compilation tool
- `--logspace`; use log space evaluation
- `-l LEAKPROB, --leak-probabilities LEAKPROB`; Add leak probabilities for evidence atoms.
- `--propagate-evidence`; Enable evidence propagation
- `--dont-propagate-evidence`; Disable evidence propagation
- `--normalize`; Normalize AD-weights.
- `-v, --verbose`;
- `-a ARGS, --arg ARGS`; Pass additional arguments to the cmd_args builtin.

An example of model file `some_heads.pl`:

```
t(_)::heads1.
t(_)::heads2.
someHeads :- heads1.
someHeads :- heads2.
```

An example of evidence file `some_heads.pl`:

```
evidence(someHeads,false).
evidence(heads1,false).
----------------
evidence(someHeads,true).
evidence(heads1,true).
----------------
evidence(someHeads,true).
evidence(heads1,false).
----------------
```

An example of LFI call:

```
$ problog lfi some_heads.pl some_heads_ev.pl -O some_heads_learned.pl
-1.7917594692732088 [0.33333333, 0.5] [t(_)::heads1, t(_)::heads2] 21
```

The learned program is saved in `some_heads_learned.pl`.

```
$ cat some_heads_learned.pl
0.33333333::heads1.
0.5::heads2.
someHeads :- heads1.
someHeads :- heads2.
```

3.5 Decision Theoretic ProbLog (dt)

Run ProbLog in decision-theoretic mode.

Used as: `problog dt <model> [optional]` where:

- `<model>` is the a decision-theoretic ProbLog model file;
- `[optional]` are optional arguments

The command standard output is `<choices> <score>` where:

- `<choices>` are the best decisions;
- `<scores>` is the score for the best decision.

The current implementation supports two evaluation strategies: exhaustive search (exact) and local search (approximate). Exhaustive search is the default. Local search can be enabled with the argument `-s local`.

The optional arguments are:

- `-h, --help`; show the help message and exit
- `--knowledge {sdd,sddx,bdd,nnf,ddnnf,kbest,fsdd,fbdd}, -k {sdd,sddx,bdd,nnf,ddnnf,kbest,fsdd,fbdd}`; Knowledge compilation tool.
- `-s {local,exhaustive}`; –search {local,exhaustive}
- `-v, --verbose`; Set verbosity level
- `-o OUTPUT, --output OUTPUT`; Write output to given file (default: write to stdout)

For example, given the DT-model:

```
$ cat dt_model.pl
0.3::rain.
0.5::wind.
?::umbrella.
?::raincoat.
broken_umbrella :- umbrella, rain, wind.
dry :- rain, raincoat.
dry :- rain, umbrella, not broken_umbrella.
dry :- not(rain).
utility(broken_umbrella, -40).
utility(raincoat, -20).
utility(umbrella, -2).
utility(dry, 60).
```

we can do:

```
$ problog dt dt_model.pl
raincoat:       0
umbrella:       1
SCORE: 43.00000000000001
```

References:

https://lirias.kuleuven.be/handle/123456789/270066

3.6 MAP inference (`map`)

Run ProbLog in MAP mode. Only facts that occur as explicit queries are assigned and all other probabilistic facts are marginalized over. MAP inference is implemented on top of DT-ProbLog.

Used as: `problog map <model> [optional]` where:

- `<model>` is the a ProbLog model file;
- `[optional]` are optional arguments

The command standard output is `<choices> <score>` where:

- `<choices>` are the MAP assignments;
- `<scores>` is the score for the MAP.

The current implementation supports two evaluation strategies: exhaustive search (exact) and local search (approximate). Exhaustive search is the default. Local search can be enabled with the argument `-s local`.

The optional arguments are:

- `-h, --help`; show the help message and exit
- `--knowledge {sdd,sddx,bdd,nnf,ddnnf,kbest,fsdd,fbdd}, -k {sdd,sddx,bdd, nnf,ddnnf,kbest,fsdd,fbdd}`; Knowledge compilation tool.
- `-s {local,exhaustive}`; –search {local,exhaustive}
- `-v, --verbose`; Set verbosity level
- `-o OUTPUT, --output OUTPUT`; Write output to given file (default: write to stdout)

3.7 Explanation mode (`explain`)

Run ProbLog in explain mode.

Used as: `problog explain <model> [optional].`

The `explain` mode offers insight in how probabilities can be computed for a ProbLog program. Given a model, the output consists of three parts:

- a reformulation of the model in which annotated disjunctions and probabilistic clauses are rewritten
- for each query, a list of mutually exclusive proofs with their probability
- for each query, the success probability determined by taking the sum of the probabilities of the individual proofs

This mode currently does not support evidence.

3.8 Grounding (`ground`)

Run ProbLog ground routine.

Used as: `problog ground <model> [optional].`

The `ground` mode provides access to the ProbLog grounder. Given a model, the output consists of the ground program.

The optional arguments are: - `-h, --help`; show the help message and exit - `--format {dot,pl,cnf,svg, internal}`; output format. The output can be formatted in different formats:

- pl: ProbLog format
- dot: GraphViz representation of the AND-OR tree
- svg: GraphViz representation of the AND-OR tree as SVG (requires GraphViz)
- cnf: DIMACS encoding as CNF
- internal: Internal representation (for debugging)
- `--break-cycles`; perform cycle breaking
- `--transform-nnf`; transform to NNF
- `--keep-all`; also output deterministic nodes
- `--keep-duplicates`; don't eliminate duplicate literals
- `--any-order`; allow reordering nodes
- `--hide-builtins`; hide deterministic part based on builtins
- `--propagate-evidence`; propagate evidence
- `--propagate-weights`; propagate evidence
- `--compact`; allow compact model (may remove some predicates)
- `--noninterpretable`;
- `--verbose, -v`; Verbose output
- `-o OUTPUT, --output OUTPUT`; output file
- `-a ARGS, --arg ARGS`; Pass additional arguments to the cmd_args builtin.

By default, the output is the ground program before cycle breaking (except for `cnf`). To perform cycle breaking, provide the `--break-cycles` argument.

3.9 Interactive shell (`shell`)

ProbLog also has an interactive shell, similar to Prolog. You can start it using the keyword `shell` as first command line argument.

The shell allows you to load models and query them interactively.

To load a file:

```
?- consult('test/3_tossing_coin.pl').
```

Queries can be specified as in Prolog:

```
?- heads(X).
X = c4,
p: 0.6;
---------------
X = c3,
p: 0.6;
---------------
X = c2,
p: 0.6;
---------------
X = c1,
```

(continues on next page)

(continued from previous page)

```
p: 0.6;
---------------
```

```
?- someHeads.
p: 0.9744;
---------------
```

Evidence can be specified using a pipe (`|`):

```
?- someHeads | not heads(c1).
```

Type `help.` for more information.

3.10 Bayesian network (`bn`)

ProbLog can export a program to a Bayesian network for comparison and verification purposes. The grounded program that is exported is defined by the query statements present in the program. The resulting network is not guaranteed to be the most efficient representation and includes additional latent variables to be able to express concepts such as annotated disjunctions. Decision nodes are not supported.

```
$ ./problog-cli.py bn some_heads.pl --format=xdsl -o some_heads.xdsl
```

The resulting file can be read by tools such as GeNIe and SMILE, BayesiaLab, Hugin or SamIam (depending on the chosen output format).

3.11 Installation (`install`)

Run the installer. This installs the SDD library. This currently only has effect on Mac OSX and Linux.

3.12 Web server (`web`)

Starts the web server.

To load libraries locally (no internet connection required), use `--local`. To open a web-browser with the editor use `--browser`.

3.13 Testing (`unittest`)

Run the unittests.

CHAPTER 4

Builtins and Libraries

ProbLog supports a subset of the Prolog language for expressing models in probabilistic logic. The main difference between ProbLog's language and Prolog is that Prolog is a complete logic programming language, whereas ProbLog is a logic representation language. This means that most of the functionality of Prolog that is related to the programming part (such as control constructs and input/output) are not supported in ProbLog.

4.1 Supported Prolog builtins

The list of supported builtins is based on Yap Prolog. See section 6 of the Yap manual for an explanation of these predicates.

In addition: ProbLog supports `consult/1` and `use_module/1`.

4.1.1 Control predicates

Supported:

- `P, Q`
- `P; Q`
- `true/0`
- `fail/0`
- `false/0`
- `\+/1`
- `not/1`
- `call/1`
- `call/N` (for N up to 9)
- `P` (alternative to call/1)

- `forall/2`

Special:

- `once/1`: In ProbLog `once/1` is an alias for `call/1`.

Not supported:

- `!/0`
- `P -> Q`
- `P *-> Q`
- `repeat`
- `incore/1` (use `call/1`)
- `call_with_args/N` (use `call/N`)
- `if(A,B,C)` (use `(A,B);(\+A,C)`)
- `ignore/1`
- `abort/0`
- `break/0`
- `halt/0`
- `halt/1`
- `catch/3`
- `throw/1`
- `garbage_collect/0`
- `garbage_collect_atoms/0`
- `gc/0`
- `nogc/0`
- `grow_heap/1`
- `grow_stack/1`

4.1.2 Handling Undefined Procedures

Alternative:

- `unknown(fail)` can be used

Not supported: all

4.1.3 Message Handling

Not supported: all

4.1.4 Predicates on Terms

Supported:

- `var/1`
- `atom/1`
- `atomic/1`
- `compound/1`
- `db_reference/1` (always fails)
- `float/1`
- `rational/1` (always fails)
- `integer/1`
- `nonvar/1`
- `number/1`
- `primitive/1`
- `simple/1`
- `callable/1`
- `ground/1`
- `arg/3`
- `functor/3`
- `T =.. L`
- `X = Y`
- `X \= Y`
- `is_list/1`
- `subsumes_term/2`

Not supported:

- `numbervars/3`
- `unify_with_occurs_check/2`
- `copy_term/2`
- `duplicate_term/2`
- `T1 =@= T2`
- `acyclic_term/1`

4.1.5 Predicates on Atoms

Not supported: all

To be added: all

4.1.6 Predicates on Characters

Not supported: all

To be added: all

4.1.7 Comparing Terms

Supported:

- `compare/3`
- `X == Y`
- `X \== Y`
- `X @< Y`
- `X @=< Y`
- `X @< Y`
- `X @> Y`
- `X @>= Y`
- `sort/2`
- `length/2` (both arguments unbound not allowed)

Not supported:

- `keysort/2`
- `predsort/2`

4.1.8 Arithmetic

Supported:

- `X`
- `-X`
- `X+Y`
- `X-Y`
- `X*Y`
- `X/Y`
- `X//Y`
- `X mod Y`
- `X rem Y` (currently same as mod)
- `X div Y`
- `exp/1`
- `log/1`
- `log10/1`

- sqrt/1
- sin/1
- cos/1
- tan/1
- asin/1
- acos/1
- atan/1
- atan/2
- sinh/1
- cosh/1
- tanh/1
- asinh/1
- acosh/1
- atanh/1
- lgamma/1
- erf/1
- erfc/1
- integer/1
- float/1
- float_fractional_part/1
- float_integer_part/1
- abs/1
- ceiling/1
- floor/1
- round/1
- sign/1
- truncate/1
- max/2
- min/2
- X ^ Y
- exp/2
- X ** Y
- X /\ Y
- X \/ Y
- X # Y
- X >< Y

- `X xor Y`
- `X << Y`
- `X >> Y`
- `\ X`
- `pi/0`
- `e/0`
- `epsilon/0`
- `inf/0`
- `nan/0`
- `X is Y`
- `X < Y`
- `X =< Y`
- `X > Y`
- `X >= Y`
- `X =:= Y`
- `X =\= Y`
- `between/3`
- `succ/2`
- `plus/3`

Not supported:

- `random/1`
- `rational/1`
- `rationalize/1`
- `gcd/2`
- `msb/1`
- `lsb/1`
- `popcount/1`
- `[X]`
- `cputime/0`
- `heapused/0`
- `local/0`
- `global/0`
- `random/0`
- `srandom/1`

4.1.9 Remaining sections

Not supported: all

4.2 ProbLog-specific builtins

- `try_call/N`: same as `call/N` but silently fail if the called predicate is undefined
- `subquery(+Goal, ?Probability)`: evaluate the Goal and return its probability
- `subquery(+Goal, +ListOfEvidence, ?Probability)`: evaluate the Goal, given the evidence, and return its Probability
- `debugprint/N`: print messages to stderr
- `write/N`: print messages to stdout
- `writenl/N`: print messages and newline to stdout
- `nl/0`: print newline to stdout
- `error/N`: raise a UserError with some message
- `cmd_args/1`: read the list of command line arguments passed to ProbLog with the '-a' arguments
- `atom_number/2`: transfrom an atom into a number
- `nocache(Functor, Arity)`: disable caching for the predicate Functor/Arity
- `numbervars/2`:
- `numbervars/3`
- `varnumbers/2`
- `subsumes_term/2`
- `subsumes_chk/2`
- `possible/1`: Perform a deterministic query on the given term.
- `clause/2`
- `clause/3`
- `create_scope/2`
- `subquery_in_scope/3`
- `subquery_in_scope/4`
- `call_in_scope/N`
- `find_scope/2`
- `set_state/1`
- `reset_state/0`
- `check_state/1`
- `print_state/0`
- `seq/1`: Unify the variable with a sequential number. Each call generates a new sequential number.

4.3 Available libraries

4.3.1 Lists

The ProbLog lists module implements all predicates from the SWI-Prolog lists library: `memberchk/2`, `member/2`, `append/3`, `append/2`, `prefix/2`, `select/3`, `selectchk/3`, `select/4`, `selectchk/4`, `nextto/3`, `delete/3`, `nth0/3`, `nth1/3`, `nth0/4`, `nth1/4`, `last/2`, `proper_length/2`, `same_length/2`, `reverse/2`, `permutation/2`, `flatten/2`, `max_member/2`, `min_member/2`, `sum_list/2`, `max_list/2`, `min_list/2`, `numlist/3`, `is_set/1`, `list_to_set/2`, `intersection/3`, `union/3`, `subset/2`, `subtract/3`.

In addition to these, the ProbLog library provides the following:

`select_uniform(+ID, +Values, ?Value, ?Rest)` ...

`select_weighted(+ID, +Weights, +Values, ?Value, ?Rest)` ...

`groupby(?List, ?Groups)` ...

`sub_list(?List, ?Before, ?Length, ?After, ?SubList)` ...

`enum_groups(+Groups, +Values, -Group, -GroupedValues)` ...

`enum_groups(+GroupValues, -Group, -GroupedValues)` ...

`unzip(ListAB,ListA,ListB)` ...

`zip(ListA,ListB,ListAB)` ...

`make_list(Len,Elem,List)` ...

4.3.2 Apply

The ProbLog lists module implements all predicates from the SWI-Prolog apply library: `include/3`, `exclude/3`, `partition/4`, `partition/5`, `maplist/2`, `maplist/3`, `maplist/4`, `maplist/5`, `convlist/3`, `foldl/4`, `foldl/5`, `foldl/6`, `foldl/7`, `scanl/4`, `scanl/5`, `scanl/6`, `scanl/7`.

4.3.3 Cut

ProbLog does not support cuts (`!`). However, it does provide the cut library to help with the modeling of ordered rulesets.

This library implements a soft cut.

1. Define a set of indexed-clauses (index is first argument)

```
r(1, a, b).
r(2, a, c).
r(3, b, c).
```

2. Call the rule using cut where you should remove the first argument

```
cut(r(A, B))
```

This will evaluate the rules in order of their index (note: NOT order in the file) and only ONE rule will match (the first one that succeeds).

e.g.:

```
cut(r(A, B)) => A = a, B = b
cut(r(a, X)) => X = b
cut(r(X, c)) => X = a
cut(r(b, X)) => X = c
```

The predicate cut/2 unifies the second argument with the Index of the matching rule.

4.3.4 Assert

The assert module allows assert and retracting facts dynamically from the internal database.

It provides the predicates `assertz/1`, `retract/1`, `retractall/1`.

4.3.5 Record

The record module allows access to non-backtrackable storage in the internal database.

It provides the predicates `current_key/1`, `recorda/2`, `recorda/3`, `recordz/2`, `recordz/3`, `erase/1`, `recorded/2`, `recorded/3`, `instance/2`.

4.3.6 Aggregate

The `aggregate` library LDL++ style of aggregation.

This functionality requires the 'aggregate' library.

```
:- use_module(library(aggregate)).
```

An aggregating clause is a clause of the form:

```
FUNCTOR(*GroupArgs, AggFunc<AggVar>) :- BODY.
```

with

- FUNCTOR: The predicate name.
- GroupArgs: (optional) list of arguments that will be used as a "group by". That is, the clause will produce a result for each distinct set
- AggFunc: An aggregation function. This can be any binary predicate that maps a list onto a term.
- AggVar: The variable over which the aggregation is computed.
- BODY: The body of the clause.

The library provides 'sum', 'avg', 'min' and 'max', but also user-defined predicates can be used.

User defined predicates have to be /2, with a list as input and some result as output. For example, the predicate proper_length/2 in lists fits this definition and can be used natively as an aggregation.

Examples

```
:- use_module(library(aggregate)).

person(a).
person(b).
person(c).
```

(continues on next page)

(continued from previous page)

```
person(d).
person(e).

salary(a, 1000).
salary(b, 1200).
salary(c, 800).
salary(d, 1100).
salary(e, 1400).

dept(a, dept_a).
dept(b, dept_a).
dept(c, dept_b).
dept(d, dept_b).
dept(e, dept_a).

% Average salary per department.
dept_salary(Dept, avg<Salary>) :- person(X), salary(X, Salary), dept(X, Dept).
query(dept_salary(Dept, Salary)).
% dept_salary(dept_a,1200.0) 1
% dept_salary(dept_b,950.0) 1

% Max salary per department.
dept_max_salary(Dept, max<Salary>) :- person(X), salary(X, Salary), dept(X, Dept).
query(dept_max_salary(Dept, Salary)).
% dept_max_salary(dept_a,1400) 1
% dept_max_salary(dept_b,1100) 1

% Average salary company-wide.
all_salary(avg<Salary>) :- person(X), salary(X, Salary), dept(X, Dept).
query(all_salary(Salary)).
% all_salary(1100.0) 1
```

These aggregates also support probabilistic data.

4.3.7 Collect

The `collect` library provides the => operator generalizing the operator `all/3`.

The general syntax of this operator is:

```
( CODEBLOCK ) => GroupBy / AggFunc(Arg1, Arg2, ..., ArgK)
```

with

- CODEBLOCK: A block of code parseable by Prolog
- AggFunc: An aggregation function to apply on the result of evaluating CODEBLOCK.
- Arg1, . . . , ArgK: An arbitrary number of arguments to the aggregation function.
- GroupBy: An optional expression over the aggregation function should be grouped.

In order to implement the aggregation operator, the user should define a predicate

```
collect_AggFunc(CodeBlock, GroupBy, Arg1, Arg2, ..., ArgK, Result)
```

Where standard aggregation function (e.g., the functions provided by the `aggregate` library) can be collected using the operator `aggregate/5` from the `aggregate` library through

```
collect_AggFunc(CodeBlock, GroupBy, AggVar, AggRes) :-
    aggregate(AggFunc, AggVar, GroupBy, CodeBlock, (GroupBy, AggRes)).
```

Considering predicates `cell(Row, Column, Value)` and `cell_type(Row, Column, Type)` we could use => to get the average per column of cell values representing an integer.

e.g.:

```
column_average(Column, Avg) :- (
   cell(Row, Column, Value),
   type(cell(Row, Column, 'int')
) => Column / avg(Value, Avg).
```

Where `collect_avg` can be defined using the operator `avg/2` from the `aggregate` library

```
collect_avg(CodeBlock, GroupBy, AggVar, AggRes) :-
   aggregate(avg, AggVar, GroupBy, CodeBlock, (GroupBy, AggRes)).
```

4.3.8 DB

The `db` library provides access to data stored in an SQLite database or a CSV-file. It provides two predicates:

`sqlite_load(+Filename)` This creates virtual predicates for each table in the database.

`sqlite_csv(+Filename, +Predicate)` This creates a new predicate for the data in the CSV file.

For a demonstration on how to use these, see this tutorial article.

4.3.9 Scope

In order to manage several Problog theories in one model, theories can be defined through the scope operator `:/2`. The left member of the scope is the scope name and its right member the predicate in the scope.

e.g.:

```
scope(1):knowledge(1).
```

Scopes can me manipulated as set of predicates.

e.g., the union of scopes can be generated through the `;/2` operator and a whole scope can be queried through the unification of its predicates:

```
scope(1):a.
scope(2):b.
scope(3):X :- scope(1):X; scope(2):X.
query(scope(3):_).

result:
 scope(3):a:   1
 scope(3):b:   1
```

The `scope` library provides additional behaviours in scopes.

Conjunction reasoning

e.g.:

```
scope(1):a.
scope(1):b.
query(scope(1):(a,b)).

result:
 scope(1):(a, b):  1
```

Temporary union through list

e.g.:

```
scope(1):a.
scope(2):b.
query([scope(1),scope(2)]:b).

result:
 [scope(1), scope(2)]:b:   1
```

All predicates outside any scope are considered in all scopes.

e.g:

```
a.
query(scope(1):a).

result:
 scope(1):a:   1
```

4.3.10 String

The `string` library provides predicates for string manipulation.

4.3.11 NLP4PLP

A library for representing and solving probability questions. See the NLP4PLP webpage for more information.

CHAPTER 5

Frequently asked questions

5.1 I get a NonGroundProbabilisticClause error. What is going on?

This error occurs when there is a successful evaluation of the body of a probabilistic clause in which not all variables have been assigned.

ProbLog requires that, after grounding, all probabilistic facts are ground (i.e. contain no variables). A simple example is

```
0.4 :: a(_).
b :- a(_).
query(b).
```

The argument of *a/1* is never instantiated.

Usually, this error will occur due to an automatic translation made by ProbLog when the model contains clauses with a probability. An example is

```
0.3::a(_,_).
0.4::c(X) :- a(X, Y).
query(c(1)).
```

This is translated internally to the program

```
a(_,_).
0.4::choice(0, 0, c(X),X,Y).
c(X) :- a(X, Y), choice(0, 0, c(X),X,Y).
query(c(1)).
```

For each probabilistic clause, a new probabilistic fact is created which contains all variables used in the clause (with a few exceptions, see below). In this case, this includes the variable Y which is never instantiated, an thus remains non-ground.

How to resolve?

There are two possible solutions:

- make sure that the variable *Y* is instantiated during grounding,
- remove variable *Y* from the clause by making a auxiliary predicate

> **Warning:** Note that these solutions are not equivalent!

The first solution can be achieved by explicitly listing the possible values of the second argument of *a/2*.

```
a(_,Y) :- Y = a; Y = b; Y = c.
0.4::c(X) :- a(X, Y).
query(c(1)).
```

Grounding this program will create three independent probabilistic facts, one for each value of *Y*. The query is true if any of these facts is true. The result of this program is therefore (1 - (1 - 0.4)^3) = **0.784**.

The second solution defines an auxiliary predicate which *hides* the variable *Y*.

```
a(_,_).
exists_a(X) :- a(X, Y).
0.4::c(X) :- exists_a(X).
query(c(1)).
```

Grounding this program creates only one probabilistic fact. The result is therefore **0.4**.

Special cases

As mentioned above, each probabilistic clause is rewritten to a regular clause and a probabilistic fact. The probabilistic fact contains all variables present in clauses except for variables that, due to the context in which they occur, will never be instantiated.

Currently, there are two such cases:

- variables uses in the pattern or goal (i.e. the first two arguments) of meta-predicates such as `findall/3` and `all/3`.
- variables only occuring within a negation

By Prolog's semantics such variables are not instantiated.

Here are some examples:

```
a(1,2).
a(1,3).
a(1,4).

0.3 :: q(X, L) :- all(Y, a(X, Y), L).

query(q(1,L)).
```

Result: q(1,[2,3,4]) with probability 0.3.

```
a(1,2).
a(1,3).
a(1,4).

my_all(A,B,C) :- all(A,B,C).

0.3 :: q(X, L) :- my_all(Y, a(X, Y), L).

query(q(1,L)).
```

Result: NonGroundProbabilisticClause: the exception only holds for the builtins `findall/3` and `all/3` and not for the custom `my_all/3`.

```
0.1::a(1,2).
0.1::a(1,3).
0.1::a(1,4).

0.3 :: q(X) :- \+ a(X, Y).

query(q(1)).
```

Result: q(1) with probability 0.2187: the variable *Y* only occurs in a negated context and is therefore ignored

```
0.1::a(1,2).
0.1::a(1,3).
0.1::a(1,4).

0.3 :: q(X) :- X \== Y, \+ a(X, Y).

query(q(1)).
```

Result: NonGroundProbabilisticClause: the variable *Y* also occurs in a normal context, but is not instantiated

```
0.1::a(1,2).
0.1::a(1,3).
0.1::a(1,4).

0.3 :: q(X) :- X \= Y, \+ a(X, Y).

query(q(1)).
```

Result: q(1) with probability 0.0: the body always fails so there is no successful evaluation

CHAPTER 6

API Documentation

6.1 problog.logic - Basic logic

This module contains basic logic constructs.

A Term can be:

- a function (see *Term*)
- a variable (see *Var*)
- a constant (see *Constant*)

Four functions are handled separately:

- conjunction (see *And*)
- disjunction (see *Or*)
- negation (see *Not*)
- clause (see *Clause*)

Syntactic sugar

Clauses can be constructed by virtue of overloading of Python operators:

Prolog	Python	English
`:-`	`<<`	clause
`,`	`&`	and
`;`	`|`	or
`\+`	`~`	not

Warning: Due to Python's operator priorities, the body of the clause has to be between parentheses.

Example:

```
from problog.logic import Var, Term

# Functors (arguments will be added later)
ancestor = Term('anc')
parent = Term('par')

# Literals
leo3 = Term('leo3')
al2 = Term('al2')
phil = Term('phil')

# Variables
X = Var('X')
Y = Var('Y')
Z = Var('Z')

# Clauses
c1 = ( ancestor(X,Y) << parent(X,Y) )
c2 = ( ancestor(X,Y) << ( parent(X,Z) & ancestor(Z,Y) ) )
c3 = ( parent( leo3, al2 ) )
c4 = ( parent( al2, phil ) )
```

term2str(*term*)

Convert a term argument to string.

Parameters **term** (*Term | None | int*) – the term to convert

Returns string representation of the given term where None is converted to '_'.

Return type str

list2term(*lst*)

Transform a Python list of terms in to a Prolog Term.

Parameters **lst** (*list of Term*) – list of Terms

Returns Term representing a Prolog list

Return type *Term*

term2list(*term*, *deep=True*)

Transform a Prolog list to a Python list of terms.

Parameters **term** (Term) – term representing a fixed length Prolog list

Raises **ValueError** – given term is not a valid fixed length Prolog list

Returns Python list containing the elements from the Prolog list

Return type list of Term

is_ground(**terms*)

Test whether a any of given terms contains a variable. :param terms: list of terms to test for the presence of variables :param terms: tuple of (Term | int | None) :return: True if none of the arguments contains any variables.

is_variable(*term*)

Test whether a Term represents a variable.

Parameters **term** – term to check

Returns True if the expression is a variable

is_list(*term*)
Test whether a Term is a list.

Parameters **term** – term to check

Returns True if the term is a list.

class Term(*functor*, **args*, ***kwdargs*)
Bases: object

A first order term, for example 'p(X,Y)'. :param functor: the functor of the term ('p' in the example) :type functor: str :param args: the arguments of the Term ('X' and 'Y' in the example) :type args: Term | None | int :param kwdargs: additional arguments; currently 'p' (probability) and 'location' (character position in input)

functor
Term functor

args
Term arguments

arity
Number of arguments

value
Value of the Term obtained by computing the function is represents

compute_value(*functions=None*)
Compute value of the Term by computing the function it represents.

Parameters **functions** – dictionary of user-defined functions

Returns value of the Term

signature
Term's signature `functor/arity`

apply(*subst*)
Apply the given substitution to the variables in the term.

Parameters **subst** (*an object with a __getitem__ method*) – A mapping from variable names to something else

Raises whatever subst.__getitem__ raises

Returns a new Term with all variables replaced by their values from the given substitution

Return type *Term*

apply_term(*subst*)
Apply the given substitution to all (sub)terms in the term.

Parameters **subst** (*an object with a __getitem__ method*) – A mapping from variable names to something else

Raises whatever subst.__getitem__ raises

Returns a new Term with all variables replaced by their values from the given substitution

Return type *Term*

with_args(**args*, ***kwdargs*)
Creates a new Term with the same functor and the given arguments.

Parameters

- **args** (*tuple of (Term | int | None)*) – new arguments for the term

- **kwdargs** (*p=Constant | p=Var | p=float*) – keyword arguments for the term

Returns a new term with the given arguments

Return type *Term*

with_probability(*p=None*)
Creates a new Term with the same functor and arguments but with a different probability.

Parameters **p** – new probability (None clears the probability)

Returns copy of the Term

is_var()
Checks whether this Term represents a variable.

is_scope_term()
Checks whether the current term is actually a term inside a scope

is_constant()
Checks whether this Term represents a constant.

is_ground()
Checks whether the term contains any variables.

is_negated()
Checks whether the term represent a negated term.

variables(*exclude_local=False*)
Extract the variables present in the term.

Returns set of variables

Return type *problog.util.OrderedSet*

class AggTerm(**args*, ***kwargs*)
Bases: *problog.logic.Term*

class Var(*name*, *location=None*, ***kwdargs*)
Bases: *problog.logic.Term*

A Term representing a variable.

Parameters **name** (str) – name of the variable

name
Name of the variable

compute_value(*functions=None*)
Compute value of the Term by computing the function it represents.

Parameters **functions** – dictionary of user-defined functions

Returns value of the Term

is_var()
Checks whether this Term represents a variable.

is_ground()
Checks whether the term contains any variables.

class Constant(*value*, *location=None*, ***kwdargs*)
Bases: *problog.logic.Term*

A constant.

Parameters **value** (str, float or int.) – the value of the constant

compute_value(*functions=None*)
Compute value of the Term by computing the function it represents.

Parameters **functions** – dictionary of user-defined functions

Returns value of the Term

is_constant()
Checks whether this Term represents a constant.

is_string()
Check whether this constant is a string.

Returns true if the value represents a string

Return type bool

is_float()
Check whether this constant is a float.

Returns true if the value represents a float

Return type bool

is_integer()
Check whether this constant is an integer.

Returns true if the value represents an integer

Return type bool

class Object(*value*, *location=None*, ***kwdargs*)
Bases: *problog.logic.Term*

A wrapped object.

Parameters **value** – the wrapped object

compute_value(*functions=None*)
Compute value of the Term by computing the function it represents.

Parameters **functions** – dictionary of user-defined functions

Returns value of the Term

is_constant()
Checks whether this Term represents a constant.

is_string()
Check whether this constant is a string.

Returns true if the value represents a string

Return type bool

is_float()
Check whether this constant is a float.

Returns true if the value represents a float

Return type bool

is_integer()
Check whether this constant is an integer.

Returns true if the value represents an integer

Return type bool

class Clause(*head*, *body*, ***kwdargs*)
Bases: *problog.logic.Term*

A clause.

class AnnotatedDisjunction(*heads*, *body*, ***kwdargs*)
Bases: *problog.logic.Term*

An annotated disjunction.

class Or(*op1*, *op2*, ***kwdargs*)
Bases: *problog.logic.Term*

classmethod from_list(*lst*)
Create a disjunction based on the terms in the list.

Parameters **lst** – list of terms

Returns disjunction over the given terms

to_list()
Extract the terms of the disjunction into the list.

Returns list of disjuncts

with_args(**args*)
Creates a new Term with the same functor and the given arguments.

Parameters

- **args** (*tuple of (Term | int | None)*) – new arguments for the term
- **kwdargs** (*p=Constant | p=Var | p=float*) – keyword arguments for the term

Returns a new term with the given arguments

Return type *Term*

class And(*op1*, *op2*, *location=None*, ***kwdargs*)
Bases: *problog.logic.Term*

classmethod from_list(*lst*)
Create a conjunction based on the terms in the list.

Parameters **lst** – list of terms

Returns conjunction over the given terms

to_list()
Extract the terms of the conjunction into the list.

Returns list of disjuncts

with_args(**args*)
Creates a new Term with the same functor and the given arguments.

Parameters

- **args** (*tuple of (Term | int | None)*) – new arguments for the term
- **kwdargs** (*p=Constant | p=Var | p=float*) – keyword arguments for the term

Returns a new term with the given arguments

Return type *Term*

class Not(*functor*, *child*, *location=None*, ***kwdargs*)
Bases: *problog.logic.Term*

is_negated()
Checks whether the term represent a negated term.

unquote(*s*)
Strip single quotes from the string.

Parameters **s** – string to remove quotes from

Returns string with quotes removed

compute_function(*func*, *args*, *extra_functions=None*)
Compute the result of an arithmetic function given by a functor and a list of arguments.

Parameters

- **func** – functor
- **args** (*(list | tuple) of (Term | int | None)*) – arguments
- **extra_functions** – additional user-defined functions

Type basestring

Raises ArithmeticError if the function is unknown or if an error occurs while computing it

Returns result of the function

Return type *Constant*

exception InstantiationError(*message*, *location=None*, ***extra*)
Bases: problog.errors.GroundingError

Error used when performing arithmetic with a non-ground term.

exception ArithmeticError(*message*, *location=None*, ***extra*)
Bases: problog.errors.GroundingError

Error used when an error occurs during evaluation of an arithmetic expression.

6.2 problog.formula - Ground programs

Data structures for propositional logic.

class BaseFormula
Bases: *problog.core.ProbLogObject*

Defines a basic logic formula consisting of nodes in some logical relation.

Each node is represented by a key. This key has the following properties:

- None indicates false
- 0 indicates true
- a number larger than 0 indicates a positive node
- the key -a with a a number larger than 0 indicates the negation of a

This data structure also support weights on nodes, names on nodes and constraints.

atomcount
Number of atoms in the formula.

get_weights()
Get weights of the atoms in the formula.

Returns dictionary of weights

Return type dict[int, *Term*]

set_weights(*weights*)
Set weights of the atoms in the formula.

Parameters **weights** (*dict[int, Term]*) – dictionary of weights

get_weight(*key*, *semiring*)
Get actual value of the node with the given key according to the given semiring.

Parameters

- **key** – key of the node (can be TRUE, FALSE or positive or negative)
- **semiring** (problog.evaluator.Semiring) – semiring to use to transform stored weight term into actual value

Returns actual value of the weight of the given node

extract_weights(*semiring*, *weights=None*)
Extracts the positive and negative weights for all atoms in the data structure.

- Atoms with weight set to neutral will get weight (semiring.one(), semiring.one()).
- If the weights argument is given, it completely replaces the formula's weights.
- All constraints are applied to the weights.
- **To specify positive and negative literal of an atom you can pass a tuple and handle it in the semiring functions po**
  (4,5)::a. 4 is the postive weight and 5 the negative.

Parameters

- **semiring** – semiring that determines the interpretation of the weights
- **weights** (*dict[(Term | int), any]*) – dictionary of { node name : weight } or { node id : weight} that overrides the builtin weights, the given weights must be in external representation.

Returns dictionary { key: (positive weight, negative weight) } where the weights are in internal representation.

Return type dict[int, tuple[any,any]]

add_name(*name*, *key*, *label=None*, *keep_name=False*)
Add a name to the given node.

Parameters

- **name** (Term) – name of the node
- **key** (*int | bool*) – key of the node
- **label** – type of label (one of LABEL_*)

get_node_by_name(*name*)
Get node corresponding to the given name.

Parameters **name** – name of the node to find

Returns key of the node

Raises KeyError if no node with the given name was found

add_query(*name*, *key*, *keep_name=False*)
Add a query name.

Same as `add_name(name, key, self.LABEL_QUERY)`.

Parameters

- **name** – name of the query
- **key** – key of the query node

add_evidence(*name*, *key*, *value*, *keep_name=False*)
Add an evidence name.

Same as `add_name(name, key, self.LABEL_EVIDENCE_???)`.

Parameters

- **name** – name of the query
- **key** – key of the query node
- **value** – value of the evidence (True, False or None)

clear_evidence()
Remove all evidence.

clear_queries()
Remove all evidence.

clear_labeled(*label*)
Remove all evidence.

get_names(*label=None*)
Get a list of all node names in the formula.

Parameters **label** – restrict to given label. If not set, all nodes are returned.

Returns list of all nodes names (of the requested type) as a list of tuples (name, key)

get_names_with_label()
Get a list of all node names in the formula with their label type.

Returns list of all nodes names with their type

queries()
Get a list of all queries.

Returns `get_names(LABEL_QUERY)`

labeled()
Get a list of all query-like labels.

Returns

evidence()
Get a list of all determined evidence. Keys are negated for negative evidence. Unspecified evidence (value None) is not included.

Returns list of tuples (name, key) for positive and negative evidence

evidence_all()
Get a list of all evidence (including undetermined).

Returns list of tuples (name, key, value) where value can be -1, 0 or 1

is_true(*key*)
Does the key represent deterministic True?

Parameters **key** – key

Returns `key == self.TRUE`

is_false(*key*)
Does the key represent deterministic False?

Parameters **key** – key

Returns `key == self.FALSE`

is_probabilistic(*key*)
Does the key represent a probabilistic node?

Parameters **key** – key

Returns `not is_true(key) and not is_false(key)`

negate(*key*)
Negate the key.

For TRUE, returns FALSE; For FALSE, returns TRUE; For x returns -x

Parameters **key** – key to negate

Returns negation of the key

constraints()
Return the list of constraints.

Returns list of constraints

add_constraint(*constraint*)
Add a constraint

Parameters **constraint** (`problog.constraint.Constraint`) – constraint to add

class atom(*identifier*, *probability*, *group*, *name*, *source*, *is_extra*)
Bases: `tuple`

group
Alias for field number 2

identifier
Alias for field number 0

is_extra
Alias for field number 5

name
Alias for field number 3

probability
Alias for field number 1

source
Alias for field number 4

class conj(*children*, *name*)
Bases: `tuple`

children
Alias for field number 0

name
Alias for field number 1

class disj(*children*, *name*)
Bases: `tuple`

children
Alias for field number 0

name
Alias for field number 1

class LogicFormula(*auto_compact=True*, *avoid_name_clash=False*, *keep_order=False*, *use_string_names=False*, *keep_all=False*, *propagate_weights=None*, *max_arity=0*, *keep_duplicates=False*, *keep_builtins=False*, *hide_builtins=False*, *database=None*, ***kwdargs*)
Bases: *problog.formula.BaseFormula*

A logic formula is a data structure that is used to represent generic And-Or graphs. It can typically contain three types of nodes:

- atom (or terminal)
- and (compound)
- or (compound)

The compound nodes contain a list of children which point to other nodes in the formula. These pointers can be positive or negative.

In addition to the basic logical structure of the formula, it also maintains a table of labels, which can be used to easily retrieve certain nodes. These labels typically contain the literals from the original program.

Upon addition of new nodes, the logic formula can perform certain optimizations, for example, by simplifying nodes or by reusing existing nodes.

add_name(*name*, *key*, *label=None*, *keep_name=False*)
Associates a name to the given node identifier.

Parameters

- **name** – name of the node
- **key** – id of the node
- **label** – type of node (see LogicFormula.LABEL_*)
- **keep_name** – keep name of node if it exists

is_trivial()
Test whether the formula contains any logical construct.

Returns False if the formula only contains atoms.

get_next_atom_identifier()
Get a unique identifier that can - and has not - been used to add a new atom. :return: A next unique identifier to use when adding new atoms (self.add_atom(identifier=..))

add_atom(*identifier*, *probability*, *group=None*, *name=None*, *source=None*, *cr_extra=True*, *is_extra=False*)
Add an atom to the formula.

Parameters

- **identifier** – a unique identifier for the atom

- **probability** – probability of the atom
- **group** – a group identifier that identifies mutually exclusive atoms (or None if no constraint)
- **name** – name of the new node
- **cr_extra** – When required, create an extra_node for the constraint group.

Returns the identifiers of the node in the formula (returns self.TRUE for deterministic atoms)

This function has the following behavior :

- If `probability` is set to `None` then the node is considered to be deterministically true and the function will return `TRUE`.
- If a node already exists with the given `identifier`, the id of that node is returned.
- If `group` is given, a mutual exclusivity constraint is added for all nodes sharing the same group.
- To add an explicitly present deterministic node you can set the probability to `True`.

add_and(*components*, *key=None*, *name=None*, *compact=None*)
Add a conjunction to the logic formula.

Parameters

- **components** – a list of node identifiers that already exist in the logic formula.
- **key** – preferred key to use
- **name** – name of the node

Returns the key of the node in the formula (returns 0 for deterministic atoms)

add_or(*components*, *key=None*, *readonly=True*, *name=None*, *placeholder=False*, *compact=None*)
Add a disjunction to the logic formula.

Parameters

- **components** – a list of node identifiers that already exist in the logic formula.
- **key** – preferred key to use
- **readonly** – indicates whether the node should be modifiable. This will allow additional disjunct to be added without changing the node key. Modifiable nodes are less optimizable.
- **name** – name of the node

Returns the key of the node in the formula (returns 0 for deterministic atoms)

Return type `int`

By default, all nodes in the data structure are immutable (i.e. readonly). This allows the data structure to optimize nodes, but it also means that cyclic formula can not be stored because the identifiers of all descendants must be known add creation time.

By setting *readonly* to False, the node is made mutable and will allow adding disjunct later using the `addDisjunct()` method. This may cause the data structure to contain superfluous nodes.

add_disjunct(*key*, *component*)
Add a component to the node with the given key.

Parameters

- **key** – id of the node to update
- **component** – the component to add

Returns key

Raises `ValueError` if `key` points to an invalid node

This may only be called with a key that points to a disjunctive node or `TRUE`.

add_not(*component*)
Returns the key to the negation of the node.

Parameters **component** – the node to negate

get_node(*key*)
Get the content of the node with the given key.

Parameters **key** (*int > 0*) – key of the node

Returns content of the node

constraints()
Returns a list of all constraints.

has_evidence_values()
Checks whether the current formula contains information for evidence propagation.

get_evidence_values()
Retrieves evidence propagation information.

get_evidence_value(*key*)
Get value of the given node based on evidence propagation.

Parameters **key** – key of the node

Returns value of the node (key, TRUE or FALSE)

set_evidence_value(*key*, *value*)
Set value of the given node based on evidence propagation.

Parameters

- **key** – key of the node
- **value** – value of the node

propagate(*nodeids*, *current=None*)
Propagate the value of the given node (true if node is positive, false if node is negative) The propagation algorithm is not complete.

Parameters

- **nodeids** – evidence nodes to set (> 0 means true, < 0 means false)
- **current** – current set of nodes with deterministic value

Returns dictionary of nodes with deterministic value

to_prolog()
Convert the Logic Formula to a Prolog program.

To make this work correctly some flags should be set on the engine and LogicFormula prior to grounding. The following code should be used:

```
pl = problog.program.PrologFile(input_file)
problog.formula.LogicFormula.create_from(pl, avoid_name_clash=True, keep_
↪order=True, label_all=True)
prologfile = gp.to_prolog()
```

Returns Prolog program

Return type str

get_name(*key*)
Get the name of the given node.

Parameters **key** – key of the node

Returns name of the node

Return type *Term*

enumerate_clauses(*relevant_only=True*)

Enumerate the clauses of this logic formula. Clauses are represented as (`head, [body]`).

Parameters **relevant_only** – only list clauses that are part of the ground program for a query or evidence

Returns iterator of clauses

to_dot(*not_as_node=True*, *nodeprops=None*)
Write out in GraphViz (dot) format.

Parameters

- **not_as_node** – represent negation as a node
- **nodeprops** – additional properties for nodes

Returns string containing dot representation

class LogicDAG(*auto_compact=True*, ***kwdargs*)
Bases: *problog.formula.LogicFormula*

A propositional logic formula without cycles.

class LogicNNF(*auto_compact=True*, ***kwdargs*)
Bases: *problog.formula.LogicDAG*, *problog.evaluator.Evaluatable*

A propositional formula in NNF form (i.e. only negation on facts).

copy_node_from(*source*, *index*, *translate=None*)
Copy a node with transformation to Negation Normal Form (only negation on facts).

class DeterministicLogicFormula(***kwdargs*)
Bases: *problog.formula.LogicFormula*

A deterministic logic formula.

add_atom(*identifier*, *probability*, *group=None*, *name=None*, *source=None*)
Add an atom to the formula.

Parameters

- **identifier** – a unique identifier for the atom
- **probability** – probability of the atom
- **group** – a group identifier that identifies mutually exclusive atoms (or None if no constraint)
- **name** – name of the new node
- **cr_extra** – When required, create an extra_node for the constraint group.

Returns the identifiers of the node in the formula (returns self.TRUE for deterministic atoms)

This function has the following behavior :

- If `probability` is set to `None` then the node is considered to be deterministically true and the function will return `TRUE`.
- If a node already exists with the given `identifier`, the id of that node is returned.
- If `group` is given, a mutual exclusivity constraint is added for all nodes sharing the same group.
- To add an explicitly present deterministic node you can set the probability to `True`.

class ClauseDB(*builtins=None*, *parent=None*)

Bases: *problog.program.LogicProgram*

Compiled logic program.

A logic program is compiled into a table of instructions. The types of instructions are:

define(functor, arity, defs) Pointer to all definitions of functor/arity. Definitions can be: `fact`, `clause` or `adc`.

clause(functor, arguments, bodynode, varcount) Single clause. Functor is the head functor, Arguments are the head arguments. Body node is a pointer to the node representing the body. Var count is the number of variables in head and body.

fact(functor, arguments, probability) Single fact.

adc(functor, arguments, bodynode, varcount, parent) Single annotated disjunction choice. Fields have same meaning as with `clause`, parent_node points to the parent `ad` node.

ad(childnodes) Annotated disjunction group. Child nodes point to the `adc` nodes of the clause.

call(functor, arguments, defnode) Body literal with call to clause or builtin. Arguments contains the call arguments, definition node is the pointer to the definition node of the given functor/arity.

conj(childnodes) Logical and. Currently, only 2 children are supported.

disj(childnodes) Logical or. Currently, only 2 children are supported.

neg(childnode) Logical not.

get_node(*index*)

Get the instruction node at the given index.

Parameters **index** (`int`) – index of the node to retrieve

Returns requested node

Return type `tuple`

Raises **IndexError** – the given index does not point to a node

find(*head*)

Find the `define` node corresponding to the given head.

Parameters **head** (`basic.Term`) – clause head to match

Returns location of the clause node in the database, returns `None` if no such node exists

Return type `int` or `None`

add_clause(*clause*, *scope=None*)

Add a clause to the database.

Parameters **clause** (`Clause`) – Clause to add

Returns location of the definition node in the database

Return type int

add_fact(*term*, *scope=None*)
Add a fact to the database. :param term: fact to add :type term: Term :return: position of the definition node in the database :rtype: int

iter_raw()
Iterate over clauses of model as represented in the database i.e. with choice facts and without annotated disjunctions.

create_function(*functor*, *arity*)
Create a Python function that can be used to query a specific predicate on this database.

Parameters

- **functor** – functor of the predicate
- **arity** – arity of the predicate (the function will take arity - 1 arguments

Returns a Python callable

exception ConsultError(*message*, *location*)
Bases: problog.errors.GroundingError

Error during consult

exception AccessError(*message*, *location=None*, ***extra*)
Bases: problog.errors.GroundingError

class ClauseIndex(*parent*, *arity*)
Bases: list

append(*item*)
Append object to the end of the list.

6.3 problog.cycles - Cycle-breaking

Cycle breaking in propositional formulae.

break_cycles(*source*, *target*, *translation=None*, *keep_named=False*, ***kwdargs*)
Break cycles in the source logic formula.

Parameters

- **source** – logic formula with cycles
- **target** – target logic formula without cycles
- **keep_named** – if true, then named nodes will be preserved after cycle breaking
- **kwdargs** – additional arguments (ignored)

Returns target

6.4 problog.constraint - Propositional constraints

Data structures for specifying propositional constraints.

class Constraint
Bases: object

A propositional constraint.

get_nodes()
Get all nodes involved in this constraint.

update_weights(*weights*, *semiring*)
Update the weights in the given dictionary according to the constraints.

Parameters

- **weights** – dictionary of weights (see result of LogicFormula.extract_weights())
- **semiring** – semiring to use for weight transformation

is_true()
Checks whether the constraint is trivially true.

is_false()
Checks whether the constraint is trivially false.

is_nontrivial()
Checks whether the constraint is non-trivial.

as_clauses()
Represent the constraint as a list of clauses (CNF form).

Returns list of clauses where each clause is represent as a list of node keys

Return type list[list[int]]

copy(*rename=None*)
Copy this constraint while applying the given node renaming.

Parameters **rename** – node rename map (or None if no rename is required)

Returns copy of the current constraint

class ConstraintAD(*group*)
Bases: *problog.constraint.Constraint*

Annotated disjunction constraint (mutually exclusive with weight update).

get_nodes()
Get all nodes involved in this constraint.

is_true()
Checks whether the constraint is trivially true.

is_false()
Checks whether the constraint is trivially false.

add(*node*, *formula*, *cr_extra=True*)
Add a node to the constraint from the given formula.

Parameters

- **node** – node to add
- **formula** – formula from which the node is taken
- **cr_extra** – Create an extra_node when required (when it is None and this is the second atom of the group).

Returns value of the node after constraint propagation

as_clauses()
Represent the constraint as a list of clauses (CNF form).

Returns list of clauses where each clause is represent as a list of node keys

Return type list[list[int]]

update_weights(*weights*, *semiring*)
Update the weights in the given dictionary according to the constraints.

Parameters

- **weights** – dictionary of weights (see result of `LogicFormula.extract_weights()`)
- **semiring** – semiring to use for weight transformation

copy(*rename=None*)
Copy this constraint while applying the given node renaming.

Parameters **rename** – node rename map (or None if no rename is required)

Returns copy of the current constraint

check(*values*)
Check the constraint

Parameters **values** – dictionary of values for nodes

Returns True if constraint succeeds, False otherwise

propagate(*values*, *weights*, *node=None*)
Returns - True: constraint satisfied - False: constraint violated - None: unknown

class ClauseConstraint(*nodes*)
Bases: *problog.constraint.Constraint*

A constraint specifying that a given clause should be true.

as_clauses()
Represent the constraint as a list of clauses (CNF form).

Returns list of clauses where each clause is represent as a list of node keys

Return type list[list[int]]

copy(*rename=None*)
Copy this constraint while applying the given node renaming.

Parameters **rename** – node rename map (or None if no rename is required)

Returns copy of the current constraint

class TrueConstraint(*node*)
Bases: *problog.constraint.Constraint*

A constraint specifying that a given node should be true.

get_nodes()
Get all nodes involved in this constraint.

as_clauses()
Represent the constraint as a list of clauses (CNF form).

Returns list of clauses where each clause is represent as a list of node keys

Return type list[list[int]]

copy(*rename=None*)
Copy this constraint while applying the given node renaming.

Parameters **rename** – node rename map (or None if no rename is required)

Returns copy of the current constraint

6.5 problog.evaluator - Commone interface for evaluation

Provides common interface for evaluation of weighted logic formulas.

exception OperationNotSupported
Bases: problog.errors.ProbLogError

class Semiring
Bases: object

Interface for weight manipulation.

A semiring is a set R equipped with two binary operations '+' and 'x'.

The semiring can use different representations for internal values and external values. For example, the Log-Probability semiring uses probabilities [0, 1] as external values and uses the logarithm of these probabilities as internal values.

Most methods take and return internal values. The execeptions are:

- value, pos_value, neg_value: transform an external value to an internal value
- result: transform an internal to an external value
- result_zero, result_one: return an external value

one()
Returns the identity element of the multiplication.

is_one(*value*)
Tests whether the given value is the identity element of the multiplication.

zero()
Returns the identity element of the addition.

is_zero(*value*)
Tests whether the given value is the identity element of the addition.

plus(*a*, *b*)
Computes the addition of the given values.

times(*a*, *b*)
Computes the multiplication of the given values.

negate(*a*)
Returns the negation. This operation is optional. For example, for probabilities return 1-a.

Raises ***OperationNotSupported*** – if the semiring does not support this operation

value(*a*)
Transform the given external value into an internal value.

result(*a*, *formula=None*)
Transform the given internal value into an external value.

normalize(*a*, *z*)
Normalizes the given value with the given normalization constant.

For example, for probabilities, returns a/z.

Raises ***OperationNotSupported*** – if z is not one and the semiring does not support this operation

pos_value(*a*, *key=None*)
Extract the positive internal value for the given external value.

neg_value(*a*, *key=None*)
Extract the negative internal value for the given external value.

result_zero()
Give the external representation of the identity element of the addition.

result_one()
Give the external representation of the identity element of the multiplication.

is_dsp()
Indicates whether this semiring requires solving a disjoint sum problem.

is_nsp()
Indicates whether this semiring requires solving a neutral sum problem.

in_domain(*a*)
Checks whether the given (internal) value is valid.

true(*key=None*)
Handle weight for deterministically true.

false(*key=None*)
Handle weight for deterministically false.

to_evidence(*pos_weight*, *neg_weight*, *sign*)
Converts the pos. and neg. weight (internal repr.) of a literal into the case where the literal is evidence. Note that the literal can be a negative atom regardless of the given sign.

Parameters

- **pos_weight** – The current positive weight of the literal.
- **neg_weight** – The current negative weight of the literal.
- **sign** – Denotes whether the literal or its negation is evidence. sign > 0 denotes the literal is evidence, otherwise its negation is evidence. Note: The literal itself can also still be a negative atom.

Returns A tuple of the positive and negative weight as if the literal was evidence. For example, for probability, returns (self.one(), self.zero()) if sign else (self.zero(), self.one())

ad_negate(*pos_weight*, *neg_weight*)
Negation in the context of an annotated disjunction. e.g. in a probabilistic context for 0.2::a ; 0.8::b, the negative label for both a and b is 1.0 such that model {a,-b} = 0.2 * 1.0 and {-a,b} = 1.0 * 0.8. For a, pos_weight would be 0.2 and neg_weight could be 0.8. The returned value is 1.0. :param pos_weight: The current positive weight of the literal (e.g. 0.2 or 0.8). Internal representation. :param neg_weight: The current negative weight of the literal (e.g. 0.8 or 0.2). Internal representation. :return: neg_weight corrected based on the given pos_weight, given the ad context (e.g. 1.0). Internal representation.

class SemiringProbability
Bases: *problog.evaluator.Semiring*

Implementation of the semiring interface for probabilities.

one()
Returns the identity element of the multiplication.

zero()
Returns the identity element of the addition.

is_one(*value*)
Tests whether the given value is the identity element of the multiplication.

is_zero(*value*)
Tests whether the given value is the identity element of the addition.

plus(*a*, *b*)
Computes the addition of the given values.

times(*a*, *b*)
Computes the multiplication of the given values.

negate(*a*)
Returns the negation. This operation is optional. For example, for probabilities return 1-a.

Raises *OperationNotSupported* – if the semiring does not support this operation

normalize(*a*, *z*)
Normalizes the given value with the given normalization constant.

For example, for probabilities, returns a/z.

Raises *OperationNotSupported* – if z is not one and the semiring does not support this operation

value(*a*)
Transform the given external value into an internal value.

is_dsp()
Indicates whether this semiring requires solving a disjoint sum problem.

in_domain(*a*)
Checks whether the given (internal) value is valid.

class SemiringLogProbability
Bases: *problog.evaluator.SemiringProbability*

Implementation of the semiring interface for probabilities with logspace calculations.

one()
Returns the identity element of the multiplication.

zero()
Returns the identity element of the addition.

is_zero(*value*)
Tests whether the given value is the identity element of the addition.

is_one(*value*)
Tests whether the given value is the identity element of the multiplication.

plus(*a*, *b*)
Computes the addition of the given values.

times(*a*, *b*)
Computes the multiplication of the given values.

negate(*a*)
Returns the negation. This operation is optional. For example, for probabilities return 1-a.

Raises *OperationNotSupported* – if the semiring does not support this operation

value(*a*)
Transform the given external value into an internal value.

result(*a*, *formula=None*)
Transform the given internal value into an external value.

normalize(*a*, *z*)
Normalizes the given value with the given normalization constant.

For example, for probabilities, returns a/z.

Raises *OperationNotSupported* – if z is not one and the semiring does not support this operation

is_dsp()
Indicates whether this semiring requires solving a disjoint sum problem.

in_domain(*a*)
Checks whether the given (internal) value is valid.

class SemiringSymbolic
Bases: *problog.evaluator.Semiring*

Implementation of the semiring interface for probabilities using symbolic calculations.

one()
Returns the identity element of the multiplication.

zero()
Returns the identity element of the addition.

plus(*a*, *b*)
Computes the addition of the given values.

times(*a*, *b*)
Computes the multiplication of the given values.

negate(*a*)
Returns the negation. This operation is optional. For example, for probabilities return 1-a.

Raises *OperationNotSupported* – if the semiring does not support this operation

value(*a*)
Transform the given external value into an internal value.

normalize(*a*, *z*)
Normalizes the given value with the given normalization constant.

For example, for probabilities, returns a/z.

Raises *OperationNotSupported* – if z is not one and the semiring does not support this operation

is_dsp()
Indicates whether this semiring requires solving a disjoint sum problem.

class Evaluatable
Bases: *problog.core.ProbLogObject*

get_evaluator(*semiring=None*, *evidence=None*, *weights=None*, *keep_evidence=False*, ***kwargs*)
Get an evaluator for computing queries on this formula. It creates an new evaluator and initializes it with the given or predefined evidence.

Parameters

- **semiring** – semiring to use
- **evidence** (*dict(Term, bool)*) – evidence values (override values defined in formula)
- **weights** – weights to use

Returns evaluator for this formula

evaluate(*index=None*, *semiring=None*, *evidence=None*, *weights=None*, ***kwargs*)
Evaluate a set of nodes.

Parameters

- **index** – node to evaluate (default: all queries)
- **semiring** – use the given semiring
- **evidence** – use the given evidence values (overrides formula)
- **weights** – use the given weights (overrides formula)

Returns The result of the evaluation expressed as an external value of the semiring. If index is `None` (all queries) then the result is a dictionary of name to value.

class EvaluatableDSP
Bases: *problog.evaluator.Evaluatable*

Interface for evaluatable formulae.

class Evaluator(*formula*, *semiring*, *weights*, ***kwargs*)
Bases: `object`

Generic evaluator.

semiring
Semiring used by this evaluator.

propagate()
Propagate changes in weight or evidence values.

evaluate(*index*)
Compute the value of the given node.

evaluate_fact(*node*)
Evaluate fact.

Parameters **node** – fact to evaluate

Returns weight of the fact (as semiring result value)

add_evidence(*node*)
Add evidence

has_evidence()
Checks whether there is active evidence.

set_evidence(*index*, *value*)
Set value for evidence node.

Parameters

- **index** – index of evidence node
- **value** – value of evidence. True if the evidence is positive, False otherwise.

set_weight(*index*, *pos*, *neg*)
Set weight of a node.

Parameters

- **index** – index of node
- **pos** – positive weight (as semiring internal value)
- **neg** – negative weight (as semiring internal value)

clear_evidence()
Clear all evidence.

evidence()
Iterate over evidence.

class FormulaEvaluator(*formula*, *semiring*, *weights=None*)
Bases: object

Standard evaluator for boolean formula.

set_weights(*weights*)
Set known weights.

Parameters **weights** – dictionary of weights

Returns

get_weight(*index*)
Get the weight of the node with the given index.

Parameters **index** – integer or formula.TRUE or formula.FALSE

Returns weight of the node

compute_weight(*index*)
Compute the weight of the node with the given index.

Parameters **index** – integer or formula.TRUE or formula.FALSE

Returns weight of the node

class FormulaEvaluatorNSP(*formula*, *semiring*, *weights=None*)
Bases: *problog.evaluator.FormulaEvaluator*

Evaluator for boolean formula that addresses the Neutral Sum Problem.

get_weight(*index*)
Get the weight of the node with the given index.

Parameters **index** – integer or formula.TRUE or formula.FALSE

Returns weight of the node and the set of abs(literals) involved

compute_weight(*index*)
Compute the weight of the node with the given index.

Parameters **index** – integer or formula.TRUE or formula.FALSE

Returns weight of the node

6.6 problog.cnf_formula - CNF

Provides access to CNF and weighted CNF.

class CNF(***kwdargs*)

Bases: *problog.formula.BaseFormula*

A logic formula in Conjunctive Normal Form.

add_atom(*atom*, *force=False*)

Add an atom to the CNF.

Parameters

- **atom** – name of the atom
- **force** – add a clause for each atom to force it's existence in the final CNF

add_comment(*comment*)

Add a comment clause.

Parameters **comment** – text of the comment

add_clause(*head*, *body*)

Add a clause to the CNF.

Parameters

- **head** – head of the clause (i.e. atom it defines)
- **body** – body of the clause

add_constraint(*constraint*, *force=False*)

Add a constraint.

Parameters

- **constraint** (problog.constraint.Constraint) – constraint to add
- **force** – force constraint to be true even though none of its values are set

to_dimacs(*partial=False*, *weighted=False*, *semiring=None*, *smart_constraints=False*, *names=False*, *invert_weights=False*)

Transform to a string in DIMACS format.

Parameters

- **partial** – split variables if possibly true / certainly true
- **weighted** – created a weighted (False, int, float)
- **semiring** – semiring for weight transformation (if weighted)
- **names** – Print names in comments

Returns string in DIMACS format

to_lp(*partial=False*, *semiring=None*, *smart_constraints=False*)

Transfrom to CPLEX lp format (MIP program). This is always weighted.

Parameters

- **partial** – split variables in possibly true / certainly true
- **semiring** – semiring for weight transformation (if weighted)
- **smart_constraints** – only enforce constraints when variables are set

Returns string in LP format

from_partial(*atoms*)

Translates a (complete) conjunction in the partial formula back to the complete formula.

For example: given an original formula with one atom '1', this atom is translated to two atoms '1' (pt) and '2' (ct).

The possible conjunctions are:

- [1, 2] => [1] certainly true (and possibly true) => true
- [-1, -2] => [-1] not possibly true (and certainly true) => false
- [1, -2] => [] possibly true but not certainly true => unknown
- [-1, 2] => INVALID certainly true but not possible => invalid (not checked)

Parameters **atoms** – complete list of atoms in partial CNF

Returns partial list of atoms in full CNF

is_trivial()
Checks whether the CNF is trivial (i.e. contains no clauses)

clauses
Return the list of clauses

clausecount
Return the number of clauses

clarks_completion(*source*, *destination*, *force_atoms=False*, ***kwdargs*)
Transform an acyclic propositional program to a CNF using Clark's completion.

Parameters

- **source** – acyclic program to transform
- **destination** – target CNF
- **kwdargs** – additional options (ignored)

Returns destination

6.7 problog.nnf_formula - d-DNNF

Provides access to d-DNNF formulae.

exception DSharpError
Bases: problog.errors.CompilationError

DSharp has crashed.

class DDNNF(***kwdargs*)
Bases: *problog.formula.LogicDAG*, *problog.evaluator.EvaluatableDSP*

A d-DNNF formula.

class SimpleDDNNFEvaluator(*formula*, *semiring*, *weights=None*, ***kwargs*)
Bases: *problog.evaluator.Evaluator*

Evaluator for d-DNNFs.

propagate()
Propagate changes in weight or evidence values.

evaluate_fact(*node*)
Evaluate fact.

Parameters **node** – fact to evaluate

Returns weight of the fact (as semiring result value)

evaluate(*node*)
Compute the value of the given node.

has_constraints(*ignore_type=None*)
Check whether the formula has any constraints that are not of the ignore_type. :param ignore_type: A set of constraint classes to ignore. :type ignore_type: None | Set

get_root_weight()
Get the WMC of the root of this formula.

Returns The WMC of the root of this formula (WMC of node len(self.formula)), multiplied with weight of True

(self.weights.get(0)).

set_weight(*index*, *pos*, *neg*)
Set weight of a node.

Parameters

- **index** – index of node
- **pos** – positive weight (as semiring internal value)
- **neg** – negative weight (as semiring internal value)

set_evidence(*index*, *value*)
Set value for evidence node.

Parameters

- **index** – index of evidence node
- **value** – value of evidence. True if the evidence is positive, False otherwise.

class Compiler
Bases: object

Interface to CNF to d-DNNF compiler tool.

classmethod get_default()
Get default compiler for this system.

classmethod get(*name*)
Get compiler by name (or default if name not found).

Parameters **name** – name of the compiler

Returns function used to call compiler

classmethod add(*name*, *func*)
Add a compiler.

Parameters

- **name** – name of the compiler
- **func** – function used to call the compiler

6.8 problog.dd_formula - Decision Diagrams

Common interface to decision diagrams (BDD, SDD).

class DD(***kwdargs*)

Bases: *problog.formula.LogicFormula*, *problog.evaluator.EvaluatableDSP*

Root class for bottom-up compiled decision diagrams.

get_manager()

Get the underlying manager

get_inode(*index*)

Get the internal node corresponding to the entry at the given index.

Parameters **index** – index of node to retrieve

Returns internal node corresponding to the given index

set_inode(*index*, *node*)

Set the internal node for the given index.

Parameters

- **index** (*int > 0*) – index at which to set the new node
- **node** – new node

get_constraint_inode()

Get the internal node representing the constraints for this formula.

build_dd()

Build the internal representation of the formula.

build_constraint_dd()

Build the internal representation of the constraint of this formula.

to_dot(**args*, ***kwargs*)

Write out in GraphViz (dot) format.

Parameters

- **not_as_node** – represent negation as a node
- **nodeprops** – additional properties for nodes

Returns string containing dot representation

class DDManager

Bases: object

Manager for decision diagrams.

add_variable(*label=0*)

Add a variable to the manager and return its label.

Parameters **label** (*int*) – suggested label of the variable

Returns label of the new variable

Return type int

literal(*label*)

Return an SDD node representing a literal.

Parameters **label** (*int*) – label of the literal

Returns internal node representing the literal

is_true(*node*)
Checks whether the SDD node represents True.

Parameters **node** – node to verify

Returns True if the node represents True

Return type bool

true()
Return an internal node representing True.

Returns

is_false(*node*)
Checks whether the internal node represents False

Parameters **node** (*SDDNode*) – node to verify

Returns False if the node represents False

Return type bool

false()
Return an internal node representing False.

conjoin2(*a*, *b*)
Base method for conjoining two internal nodes.

Parameters

- **a** – first internal node
- **b** – second internal node

Returns conjunction of given nodes

disjoin2(*a*, *b*)
Base method for disjoining two internal nodes.

Parameters

- **a** – first internal node
- **b** – second internal node

Returns disjunction of given nodes

conjoin(**nodes*)
Create the conjunction of the given nodes.

Parameters **nodes** – nodes to conjoin

Returns conjunction of the given nodes

This method handles node reference counting, that is, all intermediate results are marked for garbage collection, and the output node has a reference count greater than one. Reference count on input nodes is not touched (unless one of the inputs becomes the output).

disjoin(**nodes*)
Create the disjunction of the given nodes.

Parameters **nodes** – nodes to conjoin

Returns disjunction of the given nodes

This method handles node reference counting, that is, all intermediate results are marked for garbage collection, and the output node has a reference count greater than one. Reference count on input nodes is not touched (unless one of the inputs becomes the output).

equiv(*node1*, *node2*)
Enforce the equivalence between node1 and node2.

Parameters

- **node1** –
- **node2** –

Returns

negate(*node*)
Create the negation of the given node.

This method handles node reference counting, that is, all intermediate results are marked for garbage collection, and the output node has a reference count greater than one. Reference count on input nodes is not touched (unless one of the inputs becomes the output).

Parameters **node** – negation of the given node

Returns negation of the given node

same(*node1*, *node2*)
Checks whether two SDD nodes are equivalent.

Parameters

- **node1** – first node
- **node2** – second node

Returns True if the given nodes are equivalent, False otherwise.

Return type bool

ref(**nodes*)
Increase the reference count for the given nodes.

Parameters **nodes** (*tuple of SDDNode*) – nodes to increase count on

deref(**nodes*)
Decrease the reference count for the given nodes.

Parameters **nodes** (*tuple of SDDNode*) – nodes to decrease count on

write_to_dot(*node*, *filename*)
Write SDD node to a DOT file.

Parameters

- **node** (*SDDNode*) – SDD node to output
- **filename** (*basestring*) – filename to write to

wmc(*node*, *weights*, *semiring*)
Perform Weighted Model Count on the given node.

Parameters

- **node** – node to evaluate
- **weights** – weights for the variables in the node
- **semiring** – use the operations defined by this semiring

Returns weighted model count

wmc_literal(*node*, *weights*, *semiring*, *literal*)
Evaluate a literal in the decision diagram.

Parameters

- **node** – root of the decision diagram
- **weights** – weights for the variables in the node
- **semiring** – use the operations defined by this semiring
- **literal** – literal to evaluate

Returns weighted model count

wmc_true(*weights*, *semiring*)
Perform weighted model count on a true node. This can be used to obtain a normalization constant.

Parameters

- **weights** – weights for the variables in the node
- **semiring** – use the operations defined by this semiring

Returns weighted model count

class DDEvaluator(*formula*, *semiring*, *weights=None*, ***kwargs*)
Bases: *problog.evaluator.Evaluator*

Generic evaluator for bottom-up compiled decision diagrams.

Parameters

- **formula** –
- **semiring** –
- **weights** –

Type *DD*

Returns

propagate()
Propagate changes in weight or evidence values.

evaluate(*node*)
Compute the value of the given node.

evaluate_fact(*node*)
Evaluate fact.

Parameters **node** – fact to evaluate

Returns weight of the fact (as semiring result value)

set_evidence(*index*, *value*)
Set value for evidence node.

Parameters

- **index** – index of evidence node
- **value** – value of evidence. True if the evidence is positive, False otherwise.

set_weight(*index*, *pos*, *neg*)
Set weight of a node.

Parameters

- **index** – index of node
- **pos** – positive weight (as semiring internal value)
- **neg** – negative weight (as semiring internal value)

build_dd(*source*, *destination*, ***kwdargs*)
Build a DD from another formula.

Parameters

- **source** – source formula
- **destination** – destination formula
- **kwdargs** – extra arguments

Returns destination

6.9 problog.bdd_formula - Binary Decision Diagrams

Provides access to Binary Decision Diagrams (BDDs).

class BDD(***kwdargs*)
Bases: *problog.dd_formula.DD*

A propositional logic formula consisting of and, or, not and atoms represented as an BDD.

get_atom_from_inode(*node*)
Get the original atom given an internal node.

Parameters **node** – internal node

Returns atom represented by the internal node

classmethod is_available()
Checks whether the BDD library is available.

class BDDManager(*varcount=0*, *auto_gc=True*)
Bases: *problog.dd_formula.DDManager*

Manager for BDDs. It wraps around the pyeda BDD module

add_variable(*label=0*)
Add a variable to the manager and return its label.

Parameters **label** (*int*) – suggested label of the variable

Returns label of the new variable

Return type int

get_variable(*node*)
Get the variable represented by the given node.

Parameters **node** – internal node

Returns original node

literal(*label*)
Return an SDD node representing a literal.

Parameters **label** (*int*) – label of the literal

Returns internal node representing the literal

is_true(*node*)
Checks whether the SDD node represents True.

Parameters **node** – node to verify

Returns True if the node represents True

Return type bool

true()
Return an internal node representing True.

Returns

is_false(*node*)
Checks whether the internal node represents False

Parameters **node** (*SDDNode*) – node to verify

Returns False if the node represents False

Return type bool

false()
Return an internal node representing False.

conjoin2(*r*, *s*)
Base method for conjoining two internal nodes.

Parameters

- **a** – first internal node
- **b** – second internal node

Returns conjunction of given nodes

disjoin2(*r*, *s*)
Base method for disjoining two internal nodes.

Parameters

- **a** – first internal node
- **b** – second internal node

Returns disjunction of given nodes

negate(*node*)
Create the negation of the given node.

This method handles node reference counting, that is, all intermediate results are marked for garbage collection, and the output node has a reference count greater than one. Reference count on input nodes is not touched (unless one of the inputs becomes the output).

Parameters **node** – negation of the given node

Returns negation of the given node

same(*node1*, *node2*)
Checks whether two SDD nodes are equivalent.

Parameters

- **node1** – first node

- **node2** – second node

Returns True if the given nodes are equivalent, False otherwise.

Return type bool

ref(**nodes*)
Increase the reference count for the given nodes.

Parameters **nodes** (*tuple of SDDNode*) – nodes to increase count on

deref(**nodes*)
Decrease the reference count for the given nodes.

Parameters **nodes** (*tuple of SDDNode*) – nodes to decrease count on

write_to_dot(*node*, *filename*)
Write SDD node to a DOT file.

Parameters

- **node** (*SDDNode*) – SDD node to output
- **filename** (*basestring*) – filename to write to

wmc(*node*, *weights*, *semiring*)
Perform Weighted Model Count on the given node.

Parameters

- **node** – node to evaluate
- **weights** – weights for the variables in the node
- **semiring** – use the operations defined by this semiring

Returns weighted model count

wmc_literal(*node*, *weights*, *semiring*, *literal*)
Evaluate a literal in the decision diagram.

Parameters

- **node** – root of the decision diagram
- **weights** – weights for the variables in the node
- **semiring** – use the operations defined by this semiring
- **literal** – literal to evaluate

Returns weighted model count

wmc_true(*weights*, *semiring*)
Perform weighted model count on a true node. This can be used to obtain a normalization constant.

Parameters

- **weights** – weights for the variables in the node
- **semiring** – use the operations defined by this semiring

Returns weighted model count

build_bdd(*source*, *destination*, ***kwdargs*)
Build an SDD from another formula.

Parameters

- **source** – source formula
- **destination** – destination formula
- **kwdargs** – extra arguments

Returns destination

6.10 problog.sdd_formula - Sentential Decision Diagrams

Interface to Sentential Decision Diagrams (SDD)

class SDD(*sdd_auto_gc=False*, *var_constraint=None*, *init_varcount=-1*, ***kwdargs*)
Bases: *problog.dd_formula.DD*

A propositional logic formula consisting of and, or, not and atoms represented as an SDD.

This class has two restrictions with respect to the default LogicFormula:

- The number of atoms in the SDD should be known at construction time.
- It does not support updatable nodes.

This means that this class can not be used directly during grounding. It can be used as a target for the `makeAcyclic` method.

classmethod is_available()
Checks whether the SDD library is available.

to_internal_dot(*node=None*)
SDD for the given node, formatted for use with Graphviz dot.

Parameters **node** (*SddNode*) – The node to get the dot from.

Returns The dot format of the given node. When node is None, the shared_sdd will be used (contains all active

sdd structures). :rtype: str

sdd_to_dot(*node*, *litnamemap=None*, *show_id=False*, *merge_leafs=False*)
SDD for the given node, formatted for use with Graphviz dot. This method provides more control over the used symbols than to_internal_dot (see litnamemap). Primes are given by a dotted line, subs by a full line.

Parameters

- **node** (*SddNode*) – The node to get the dot from.
- **litnamemap** (*dict[(int | str), str] | bool | None*) – A dictionary providing the symbols to use. The following options are available: 1. literals, e.g. {1:'A', -1:'-A', ...}, 2. True/False, e.g. {true':'1', 'false':'0'} 3. And/Or e.g. {'mult':'x', 'add':'+'} When litnamemap = True, self.get_litnamemap() will be used.
- **show_id** – Whether to display the ids of each sdd node.
- **merge_leafs** – Whether to merge the same leaf nodes. True results in less nodes but makes it harder to

render without having crossing lines. :return: The dot format of the given node. When node is None, this mgr is used instead. :rtype: str

get_litnamemap()
Get a dictionary mapping literal IDs (inode index) to names. e.g; {1:'x', -1:'-x'}

to_formula()
Extracts a LogicFormula from the SDD.

class SDDManager(*varcount=0*, *auto_gc=False*, *var_constraint=None*)
Bases: *problog.dd_formula.DDManager*

Manager for SDDs. It wraps around the SDD library and offers some additional methods.

get_manager()
Get the underlying sdd manager.

add_variable(*label=0*)
Add a variable to the manager and return its label.

Parameters **label** (*int*) – suggested label of the variable

Returns label of the new variable

Return type int

literal(*label*)
Return an SDD node representing a literal.

Parameters **label** (*int*) – label of the literal

Returns internal node representing the literal

is_true(*node*)
Checks whether the SDD node represents True.

Parameters **node** – node to verify

Returns True if the node represents True

Return type bool

true()
Return an internal node representing True.

Returns

is_false(*node*)
Checks whether the internal node represents False

Parameters **node** (*SDDNode*) – node to verify

Returns False if the node represents False

Return type bool

false()
Return an internal node representing False.

conjoin2(*a*, *b*)
Base method for conjoining two internal nodes.

Parameters

- **a** – first internal node
- **b** – second internal node

Returns conjunction of given nodes

disjoin2(*a*, *b*)
Base method for disjoining two internal nodes.

Parameters

- **a** – first internal node
- **b** – second internal node

Returns disjunction of given nodes

negate(*node*)
Create the negation of the given node.

This method handles node reference counting, that is, all intermediate results are marked for garbage collection, and the output node has a reference count greater than one. Reference count on input nodes is not touched (unless one of the inputs becomes the output).

Parameters **node** – negation of the given node

Returns negation of the given node

same(*node1*, *node2*)
Checks whether two SDD nodes are equivalent.

Parameters

- **node1** – first node
- **node2** – second node

Returns True if the given nodes are equivalent, False otherwise.

Return type bool

ref(**nodes*)
Increase the reference count for the given nodes.

Parameters **nodes** (*tuple of SDDNode*) – nodes to increase count on

deref(**nodes*)
Decrease the reference count for the given nodes.

Parameters **nodes** (*tuple of SDDNode*) – nodes to decrease count on

write_to_dot(*node*, *filename*, *litnamemap=None*)
Write SDD node to a DOT file.

Parameters

- **node** (*SDDNode*) – SDD node to output
- **filename** (*basestring*) – filename to write to

to_internal_dot(*node=None*)
SDD for the given node, formatted for use with Graphviz dot.

Parameters **node** (*SddNode*) – The node to get the dot from.

Returns The dot format of the given node. When node is None, the shared_sdd will be used (contains all active

sdd structures). :rtype: str

sdd_to_dot(*node*, *litnamemap=None*, *show_id=False*, *merge_leafs=False*)
SDD for the given node, formatted for use with Graphviz dot. This method provides more control over the used symbols than to_internal_dot (see litnamemap). Primes are given by a dotted line, subs by a full line.

Parameters

- **node** – The node to get the dot from.

- **litnamemap** (*dict[(int | str), str] | None*) – A dictionary providing the symbols to use. The following options are available: 1. literals, e.g. {1:'A', -1:'-A', ...}, 2. True/False, e.g. {true':'1', 'false':'0'} 3. And/Or e.g. {'mult':'x', 'add':'+'}
- **show_id** – Whether to display the ids of each sdd node.
- **merge_leafs** – Whether to merge the same leaf nodes. True results in less nodes but makes it harder to

render without having crossing lines. :return: The dot format of the given node. When node is None, the mgr is used (this behavior can be overriden). :rtype: str

wmc(*node*, *weights*, *semiring*, *literal=None*, *pr_semiring=True*, *perform_smoothing=True*, *smooth_to_root=False*, *wmc_func=None*)
Perform Weighted Model Count on the given node or the given literal.

Common usage: wmc(node, weights, semiring) and wmc(node, weights, semiring, smooth_to_root=True)

Parameters

- **node** – node to evaluate Type: SddNode
- **weights** (*dict[int, tuple[Any, Any]]*) – weights for the variables in the node. Type: {literal_id : (pos_weight, neg_weight)}
- **semiring** (*Semiring*) – use the operations defined by this semiring. Type: Semiring
- **literal** – When a literal is given, the result of WMC(literal) is returned instead.
- **pr_semiring** (*bool*) – Whether the given semiring is a (logspace) probability semiring.
- **perform_smoothing** (*bool*) – Whether to perform smoothing. When pr_semiring is True, smoothing is performed regardless.
- **smooth_to_root** (*bool*) – Whether to perform smoothing compared to the root. When pr_semiring is True, smoothing compared to the root is not performed regardless of this flag.
- **wmc_func** (*function*) – The WMC function to use. If None, a built_in one will be used that depends on the given semiring. Type: function[SddNode, List[Tuple[prime_weight, sub_weight, Set[prime_used_lit], Set[sub_used_lit]]], Set[expected_prime_lit], Set[expected_sub_lit]] -> weight

Returns weighted model count of node if literal=None, else the weights are propagated up to node but the weighted model count of literal is returned.

wmc_literal(*node*, *weights*, *semiring*, *literal*)
Evaluate a literal in the decision diagram.

Parameters

- **node** – root of the decision diagram
- **weights** – weights for the variables in the node
- **semiring** – use the operations defined by this semiring
- **literal** – literal to evaluate

Returns weighted model count

wmc_true(*weights*, *semiring*)
Perform weighted model count on a true node. This can be used to obtain a normalization constant.

Parameters

- **weights** – weights for the variables in the node
- **semiring** – use the operations defined by this semiring

Returns weighted model count

get_deepcopy_noref()
Get a deep copy of this without reference counts to inodes. Notes: No inode will have a reference count and auto_gc_and_minimize will be disabled. :return: A deep copy of this without any reference counts.

class x_constrained(*X*)
Bases: tuple

X
Alias for field number 0

class SDDEvaluator(*formula*, *semiring*, *weights=None*, ***kwargs*)
Bases: *problog.dd_formula.DDEvaluator*

build_sdd(*source*, *destination*, ***kwdargs*)
Build an SDD from another formula.

Parameters

- **source** (LogicDAG) – source formula
- **destination** (SDD) – destination formula
- **kwdargs** – extra arguments

Returns destination

6.11 problog.core - Binary Decision Diagrams

Provides core functionality of ProbLog.

class ProbLog
Bases: object

Static class containing transformation information

classmethod register_transformation(*src*, *target*, *action=None*)
Register a transformation from class src to class target using function action.

Parameters

- **src** – source function
- **target** – target function
- **action** – transformation function

classmethod register_create_as(*repl*, *orig*)
Register that we can create objects of class *repl* in the same way as objects of class *orig*.

Parameters

- **repl** – object we want to create
- **orig** – object construction we can use instead

classmethod register_allow_subclass(*orig*)
Register that we can create objects of class *repl* by creating an object of a subclass.

Parameters **orig** –

classmethod find_paths(*src*, *target*, *stack=()*)
Find all possible paths to transform the src object into the target class.

Parameters

- **src** – object to transform
- **target** – class to tranform the object to
- **stack** – stack of intermediate classes

Returns list of class, action, class, action, ..., class

classmethod convert(*src*, *target*, ***kwdargs*)
Convert the source object into an object of the target class.

Parameters

- **src** – source object
- **target** – target class
- **kwdargs** – additional arguments passed to transformation functions

exception TransformationUnavailable
Bases: Exception

Exception thrown when no valid transformation between two ProbLogObjects can be found.

class ProbLogObject
Bases: object

Root class for all convertible objects in the ProbLog system.

classmethod create_from(*obj*, ***kwdargs*)
Transform the given object into an object of the current class using transformations.

Parameters

- **obj** – obj to transform
- **kwdargs** – additional options

Returns object of current class

classmethod createFrom(*obj*, ***kwdargs*)
Transform the given object into an object of the current class using transformations.

Parameters

- **obj** – obj to transform
- **kwdargs** – additional options

Returns object of current class

classmethod create_from_default_action(*src*)
Create object of this class from given source object using default action.

Parameters **src** – source object to transform

Returns transformed object

transform_create_as(*cls1*, *cls2*)
Informs the system that cls1 can be used instead of cls2 in any transformations.

Parameters

- **cls1** –

- **cls2** –

Returns

class transform(*cls1*, *cls2*, *func=None*)
Bases: object

Decorator for registering a transformation between two classes.

Parameters

- **cls1** – source class
- **cls2** – target class
- **func** – transformation function (for direct use instead of decorator)

list_transformations()
Print an overview of available transformations.

6.12 problog.engine - Grounding engine

Grounding engine to transform a ProbLog program into a propositional formula.

ground(*model*, *target=None*, *grounder=None*, ***kwdargs*)
Ground a given model.

Parameters **model** (LogicProgram) – logic program to ground

Returns the ground program

Return type *LogicFormula*

ground_default(*model*, *target=None*, *queries=None*, *evidence=None*, *propagate_evidence=False*, *labels=None*, *engine=None*, ***kwdargs*)
Ground a given model.

Parameters

- **model** (LogicProgram) – logic program to ground
- **target** (LogicFormula) – formula in which to store ground program
- **queries** – list of queries to override the default
- **evidence** – list of evidence atoms to override the default

Returns the ground program

Return type *LogicFormula*

class GenericEngine
Bases: object

Generic interface to a grounding engine.

prepare(*db: problog.program.LogicProgram*)
Prepare the given database for querying. Calling this method is optional.

Parameters **db** – logic program

Returns logic program in optimized format where builtins are initialized and directives have been evaluated

query(*db: problog.program.LogicProgram*, *term*)
Evaluate a query without generating a ground program.

Parameters

- **db** – logic program
- **term** – term to query; variables should be represented as None

Returns list of tuples of argument for which the query succeeds.

ground(*db: problog.program.LogicProgram*, *term*, *target=None*, *label=None*)
Ground a given query term and store the result in the given ground program.

Parameters

- **db** – logic program
- **term** – term to ground; variables should be represented as None
- **target** – target logic formula to store grounding in (a new one is created if none is given)
- **label** – optional label (query, evidence, . . .)

Returns logic formula (target if given)

ground_all(*db: problog.program.LogicProgram*, *target=None*, *queries=None*, *evidence=None*)
Ground all queries and evidence found in the the given database.

Parameters

- **db** – logic program
- **target** – logic formula to ground into
- **queries** – list of queries to evaluate instead of the ones in the logic program
- **evidence** – list of evidence to evaluate instead of the ones in the logic program

Returns ground program

class ClauseDBEngine(*builtins=True*, ***kwdargs*)
Bases: *problog.engine.GenericEngine*

Parent class for all Python ClauseDB-based engines.

load_builtins()
Load default builtins.

get_builtin(*index*)
Get builtin's evaluation function based on its identifier. :param index: index of the builtin :return: function that evaluates the builtin

add_builtin(*predicate*, *arity*, *function*)
Add a builtin.

Parameters

- **predicate** – name of builtin predicate
- **arity** – arity of builtin predicate
- **function** – function to execute builtin

get_builtins()
Get the list of builtins.

prepare(*db*)
Convert given logic program to suitable format for this engine. Calling this method is optional, but it allows to perform multiple operations on the same database. This also executes any directives in the input model.

Parameters **db** – logic program to prepare for evaluation

Returns logic program in a suitable format for this engine

Return type *ClauseDB*

get_non_cache_functor()
Get a unique functor that is excluded from caching.

Returns unique functor that is excluded from caching

Return type basestring

create_context(*content*, *define=None*, *parent=None*)
Create a variable context.

query(*db*, *term*, *backend=None*, ***kwdargs*)

Parameters

- **db** –
- **term** –
- **kwdargs** –

Returns

ground(*db*, *term*, *target=None*, *label=None*, ***kwdargs*)
Ground a query on the given database.

Parameters

- **db** (LogicProgram) – logic program
- **term** (Term) – query term
- **label** (*str*) – type of query (e.g. query, evidence or -evidence)
- **kwdargs** – additional arguments

Returns ground program containing the query

Return type *LogicFormula*

ground_step(*db*, *term*, *gp=None*, *silent_fail=True*, *assume_prepared=False*, ***kwdargs*)

Parameters

- **db** (LogicProgram) –
- **term** –
- **gp** –
- **silent_fail** –
- **assume_prepared** –
- **kwdargs** –

Returns

ground_all(*db*, *target=None*, *queries=None*, *evidence=None*, *propagate_evidence=False*, *labels=None*)
Ground all queries and evidence found in the the given database.

Parameters

- **db** – logic program
- **target** – logic formula to ground into
- **queries** – list of queries to evaluate instead of the ones in the logic program
- **evidence** – list of evidence to evaluate instead of the ones in the logic program

Returns ground program

exception UnknownClauseInternal
Bases: Exception

Undefined clause in call used internally.

exception NonGroundProbabilisticClause(*location*)
Bases: problog.errors.GroundingError

Encountered a non-ground probabilistic clause.

exception UnknownClause(*signature*, *location*)
Bases: problog.errors.GroundingError

Undefined clause in call.

6.13 problog.engine_builtin - Grounding engine builtins

Implementation of Prolog / ProbLog builtins.

add_standard_builtins(*engine*, *b=None*, *s=None*, *sp=None*)
Adds standard builtins to the given engine.

Parameters

- **engine** (ClauseDBEngine) – engine to add builtins to
- **b** – wrapper for boolean builtins (returning True/False)
- **s** – wrapper for simple builtins (return deterministic results)
- **sp** – wrapper for probabilistic builtins (return probabilistic results)

exception CallModeError(*functor*, *args*, *accepted=None*, *message=None*, *location=None*)
Bases: problog.errors.GroundingError

Represents an error in builtin argument types.

class StructSort(*obj*, **args*)
Bases: object

Comparator of terms based on structure.

check_mode(*args*, *accepted*, *functor=None*, *location=None*, *database=None*, ***kwdargs*)
Checks the arguments against a list of accepted types.

Parameters

- **args** (*tuple of Term*) – arguments to check

- **accepted** (*list of str*) – list of accepted combination of types (see mode_types)
- **functor** – functor of the call (used for error message)
- **location** – location of the call (used for error message)
- **database** – database (used for error message)
- **kwdargs** – additional arguments (not used)

Returns the index of the first mode in accepted that matches the arguments

Return type int

list_elements (*term*)
Extract elements from a List term. Ignores the list tail.

Parameters **term** (Term) – term representing a list

Returns elements of the list

Return type list of Term

list_tail (*term*)
Extract the tail of the list.

Parameters **term** (Term) – Term representing a list

Returns tail of the list

Return type *Term*

exception IndirectCallCycleError (*location=None*)
Bases: problog.errors.GroundingError

Cycle should not pass through indirect calls (e.g. call/1, findall/3).

6.14 problog.engine_stack - Stack-based implementation of grounding engine

Default implementation of the ProbLog grounding engine.

exception InvalidEngineState
Bases: Exception

class StackBasedEngine (*label_all=False*, ***kwdargs*)
Bases: *problog.engine.ClauseDBEngine*

load_builtins ()
Load default builtins.

in_cycle (*pointer*)
Check whether the node at the given pointer is inside a cycle.

Parameters **pointer** –

Returns

execute (*node_id*, *target=None*, *database=None*, *subcall=False*, *is_root=False*, *name=None*, ***kwdargs*)
Execute the given node. :param node_id: pointer of the node in the database :param subcall: indicates whether this is a toplevel call or a subcall :param target: target datastructure for storing the ground program

:param database: database containing the logic program to ground :param kwdargs: additional arguments :return: results of the execution

cleanup(*obj*)
Remove the given node from the stack and lower the pointer. :param obj: pointer of the object to remove :type obj: int

create_context(*content*, *define=None*, *parent=None*, *state=None*)
Create a variable context.

class State(**args*, ***kwargs*)
Bases: dict

class Context(*parent*, *state=None*)
Bases: list

class FixedContext
Bases: tuple

class MessageQueue
Bases: object

A queue of messages.

append(*message*)
Add a message to the queue.

Parameters **message** –

Returns

cycle_exhausted()
Check whether there are messages inside the cycle.

Returns

pop()
Pop a message from the queue.

Returns

class MessageFIFO(*engine*)
Bases: *problog.engine_stack.MessageQueue*

append(*message*)
Add a message to the queue.

Parameters **message** –

Returns

pop()
Pop a message from the queue.

Returns

cycle_exhausted()
Check whether there are messages inside the cycle.

Returns

class MessageAnyOrder(*engine*)
Bases: *problog.engine_stack.MessageQueue*

cycle_exhausted()
Check whether there are messages inside the cycle.

Returns

class MessageOrderD(*engine*)
Bases: *problog.engine_stack.MessageAnyOrder*

append(*message*)
Add a message to the queue.

Parameters **message** –

Returns

pop()
Pop a message from the queue.

Returns

class MessageOrderDrc(*engine*)
Bases: *problog.engine_stack.MessageAnyOrder*

append(*message*)
Add a message to the queue.

Parameters **message** –

Returns

pop()
Pop a message from the queue.

Returns

class BooleanBuiltIn(*base_function*)
Bases: object

Simple builtin that consist of a check without unification. (e.g. var(X), integer(X), ...).

class SimpleBuiltIn(*base_function*)
Bases: object

Simple builtin that does cannot be involved in a cycle or require engine information and has 0 or more results.

class SimpleProbabilisticBuiltIn(*base_function*)
Bases: object

Simple builtin that does cannot be involved in a cycle or require engine information and has 0 or more results.

class MessageOrder1(*engine*)
Bases: *problog.engine_stack.MessageAnyOrder*

append(*message*)
Add a message to the queue.

Parameters **message** –

Returns

pop()
Pop a message from the queue.

Returns

6.15 problog.engine_unify - Unification

Implementation of unification for the grounding engine.

exception UnifyError
Bases: Exception

Unification error (used and handled internally).

substitute_all(*terms*, *subst*, *wrapped=False*)

Parameters

- **terms** –
- **subst** –

Returns

instantiate(*term*, *context*)
Replace variables in Term by values based on context lookup table.

Parameters

- **term** –
- **context** –

Returns

exception OccursCheck(*location=None*)
Bases: problog.errors.GroundingError

unify_value(*value1*, *value2*, *source_values*)
Unify two values that exist in the same context. :param value1: :param value2: :param source_values: :return:

unify_value_dc(*value1*, *value2*, *source_values*, *target_values*)
Unify two values that exist in different contexts. Updates the mapping of variables from value1 to values from value2.

Parameters

- **value1** –
- **value2** –
- **source_values** – mapping of source variable to target value
- **target_values** – mapping of target variable to TARGET value

substitute_call_args(*terms*, *context*, *min_var*)

Parameters

- **terms** –
- **context** –

Returns

substitute_head_args(*terms*, *context*)
Extract the clause head arguments from the clause context. :param terms: head arguments. These can contain variables >0. :param context: clause context. These can contain variable <0. :return: input terms where variables are substituted by their values in the context

substitute_simple(*term*, *context*)

Parameters

- **term** –
- **context** –

Returns

unify_call_head(*call_args*, *head_args*, *target_context*)
Unify argument list from clause call and clause head. :param call_args: arguments of the call :param head_args: arguments of the head :param target_context: list of values of variables in the clause :raise UnifyError: unification failed

unify_call_return(*result*, *call_args*, *context*, *var_translate*, *min_var*, *mask=None*)
Transforms the result returned by a call into the calling context.

Parameters

- **result** – result returned by call
- **call_args** – arguments used in the call
- **context** – calling context
- **var_translate** – variable translation for local variables from call context to calling context
- **min_var** – number of local variables currently in calling context
- **mask** – mask indicating whether call_args are non-ground (ground can be skipped in unification)

6.16 problog.extern - Calling Python from ProbLog

Interface for calling Python from ProbLog.

6.17 problog.forward - Forward compilation and evaluation

Forward compilation using TP-operator.

class ForwardInference(*compile_timeout=None*, ***kwdargs*)
Bases: *problog.dd_formula.DD*

build_dd()
Build the internal representation of the formula.

update_inode(*index*)
Recompute the inode at the given index.

get_inode(*index*, *final=False*)
Get the internal node corresponding to the entry at the given index. :param index: index of node to retrieve :return: SDD node corresponding to the given index :rtype: SDDNode

set_inode(*index*, *node*)
Set the internal node for the given index.

Parameters

- **index** (*int > 0*) – index at which to set the new node
- **node** – new node

add_constraint(*c*)
Add a constraint

Parameters **constraint** (problog.constraint.Constraint) – constraint to add

class ForwardSDD(***kwargs*)
Bases: *problog.formula.LogicFormula*, *problog.evaluator.EvaluatableDSP*

class ForwardBDD(***kwargs*)
Bases: *problog.formula.LogicFormula*, *problog.evaluator.EvaluatableDSP*

class ForwardEvaluator(*formula*, *semiring*, *fdd*, *weights=None*, *verbose=None*, ***kwargs*)
Bases: *problog.evaluator.Evaluator*

An evaluator using anytime forward compilation.

propagate()
Propagate changes in weight or evidence values.

evaluate(*index*)
Compute the value of the given node.

has_evidence()
Checks whether there is active evidence.

clear_evidence()
Clear all evidence.

evidence()
Iterate over evidence.

6.18 problog.kbest - K-Best inference using MaxSat

Anytime evaluation using best proofs.

class KBestFormula(***kwargs*)
Bases: *problog.cnf_formula.CNF*, *problog.evaluator.Evaluatable*

classmethod is_available()
Checks whether the SDD library is available.

class KBestEvaluator(*formula*, *semiring*, *weights=None*, *lower_only=False*, *verbose=None*, *convergence=1e-09*, *explain=None*, ***kwargs*)
Bases: *problog.evaluator.Evaluator*

propagate()
Propagate changes in weight or evidence values.

evaluate(*index*)
Compute the value of the given node.

add_evidence(*node*)
Add evidence

has_evidence()
Checks whether there is active evidence.

clear_evidence()
Clear all evidence.

evidence()
Iterate over evidence.

6.19 problog.maxsat - Interface to MaxSAT solvers

Interface to MaxSAT solvers.

exception UnsatisfiableError
Bases: problog.errors.ProbLogError

6.20 problog.parser - Parser for Prolog programs

Efficient low-level parser for Prolog programs.

exception ParseError(*string*, *message*, *location*)
Bases: problog.errors.ParseError

exception UnexpectedCharacter(*string*, *position*)
Bases: *problog.parser.ParseError*

exception UnmatchedCharacter(*string*, *position*, *length=1*)
Bases: *problog.parser.ParseError*

class Factory
Bases: object

Factory object for creating suitable objects from the parse tree.

6.21 problog.program - Representation of Logic Programs

Provides tools for loading logic programs.

class LogicProgram(*source_root='.'*, *source_files=None*, *line_info=None*, ***extra_info*)
Bases: *problog.core.ProbLogObject*

add_clause(*clause*, *scope=None*)
Add a clause to the logic program.

Parameters **clause** – add a clause

add_fact(*fact*, *scope=None*)
Add a fact to the logic program.

Parameters **fact** – add a fact

classmethod create_from(*src*, *force_copy=False*, ***extra*)
Create a LogicProgram of the current class from another LogicProgram.

Parameters

- **src** (*LogicProgram*) – logic program to convert
- **force_copy** (*bool*) – default False, If true, always create a copy of the original logic program.
- **extra** – additional arguments passed to all constructors and action functions

Returns LogicProgram that is (externally) identical to given one

Return type object of the class on which this method is invoked

If the original LogicProgram already has the right class and force_copy is False, then the original program is returned.

classmethod createFrom(*src*, *force_copy=False*, ***extra*)
Create a LogicProgram of the current class from another LogicProgram.

Parameters

- **src** (*LogicProgram*) – logic program to convert
- **force_copy** (*bool*) – default False, If true, always create a copy of the original logic program.
- **extra** – additional arguments passed to all constructors and action functions

Returns LogicProgram that is (externally) identical to given one

Return type object of the class on which this method is invoked

If the original LogicProgram already has the right class and force_copy is False, then the original program is returned.

lineno(*char*, *force_filename=False*)
Transform character position to line:column format.

Parameters

- **char** – character position
- **force_filename** – always add filename even for top-level file

Returns line, column (or None if information is not available)

class SimpleProgram
Bases: *problog.program.LogicProgram*

LogicProgram implementation as a list of clauses.

add_clause(*clause*, *scope=None*)
Add a clause to the logic program.

Parameters **clause** – add a clause

add_fact(*fact*, *scope=None*)
Add a fact to the logic program.

Parameters **fact** – add a fact

class PrologString(*string*, *parser=None*, *factory=None*, *source_root='.'*, *source_files=None*, *identifier=0*)
Bases: *problog.program.LogicProgram*

Read a logic program from a string of ProbLog code.

add_clause(*clause*, *scope=None*)
Add a clause to the logic program.

Parameters **clause** – add a clause

add_fact(*fact*, *scope=None*)
Add a fact to the logic program.

Parameters **fact** – add a fact

class PrologFile(*filename*, *parser=None*, *factory=None*, *identifier=0*)
Bases: *problog.program.PrologString*

LogicProgram implementation as a pointer to a Prolog file.

Parameters

- **filename** (*string*) – filename of the Prolog file (optional)
- **identifier** – index of the file (in case of multiple files)

add_clause(*clause*, *scope=None*)
Add a clause to the logic program.

Parameters **clause** – add a clause

add_fact(*fact*, *scope=None*)
Add a fact to the logic program.

Parameters **fact** – add a fact

class PrologFactory(*identifier=0*)
Bases: *problog.parser.Factory*

Factory object for creating suitable objects from the parse tree.

class ExtendedPrologFactory(*identifier=0*)
Bases: *problog.program.PrologFactory*

Prolog with some extra syntactic sugar.

Non-standard syntax: - Negative head literals [Meert and Vennekens, PGM 2014]: 0.5:: +a :- b.

build_program(*clauses*)
Update functor f that appear as a negative head literal to f_p and :param clauses: :return:

neg_head_literal_to_pos_literal(*literal*)
Translate a negated literal into a positive literal and remember the literal to update the complete program later (in build_program). :param literal: :return:

build_probabilistic(*operand1*, *operand2*, *location=None*, ***extra*)
Detect probabilistic negated head literal and translate to positive literal :param operand1: :param operand2: :param location: :param extra: :return:

build_clause(*functor*, *operand1*, *operand2*, *location=None*, ***extra*)
Detect deterministic head literal and translate to positive literal :param functor: :param operand1: :param operand2: :param location: :param extra: :return:

DefaultPrologFactory
alias of *problog.program.ExtendedPrologFactory*

6.22 problog.setup - Installation tools

Provides an installer for ProbLog dependencies.

set_environment()
Updates local PATH and PYTHONPATH to include additional component directories.

get_binary_paths()
Get a list of additional binary search paths.

get_module_paths()
Get a list of additional module search paths.

gather_info()
Collect info about the system and its installed software.

6.23 problog.util - Useful utilities

Provides useful utilities functions and classes.

class ProbLogLogFormatter
Bases: logging.Formatter

format(*message*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

init_logger(*verbose=None*, *name='problog'*, *out=None*)
Initialize default logger.

Parameters

- **verbose** (*int*) – verbosity level (0: WARNING, 1: INFO, 2: DEBUG)
- **name** (*str*) – name of the logger (default: problog)

Returns result of `logging.getLogger(name)`

Return type logging.Logger

class Timer(*msg*, *output=None*, *logger='problog'*)
Bases: object

Report timing information for a block of code. To be used as a `with` block.

Parameters

- **msg** (*str*) – message to print
- **output** (*file*) – file object to write to (default: write to logger `problog`)

start_timer(*timeout=0*)
Start a global timeout timer.

Parameters **timeout** (*int*) – timeout in seconds

stop_timer()
Stop the global timeout timer.

subprocess_check_output(**popenargs*, ***kwargs*)
Wrapper for subprocess.check_output that recursively kills subprocesses when Python is interrupted.

Additionally expands executable name to full path.

Parameters

- **popenargs** – positional arguments of subprocess.call
- **kwargs** – keyword arguments of subprocess.call

Returns result of subprocess.call

subprocess_check_call(**popenargs*, ***kwargs*)
Wrapper for subprocess.check_call that recursively kills subprocesses when Python is interrupted.

Additionally expands executable name to full path.

Parameters

- **popenargs** – positional arguments of subprocess.call
- **kwargs** – keyword arguments of subprocess.call

Returns result of subprocess.call

subprocess_call(**popenargs*, ***kwargs*)

Wrapper for subprocess.call that recursively kills subprocesses when Python is interrupted.

Additionally expands executable name to full path.

Parameters

- **popenargs** – positional arguments of subprocess.call
- **kwargs** – keyword arguments of subprocess.call

Returns result of subprocess.call

kill_proc_tree(*process*, *including_parent=True*)

Recursively kill a subprocess. Useful when the subprocess is a script. Requires psutil but silently fails when it is not present.

Parameters

- **process** (*subprocess.Popen*) – process
- **including_parent** (*bool*) – also kill process itself (default: True)

class OrderedSet(*iterable=None*)

Bases: collections.abc.MutableSet

Provides an ordered version of a set which keeps elements in the order they are added.

Parameters **iterable** (*Sequence*) – add elements from this iterable (default: None)

add(*key*)

Add element.

Parameters **key** – element to add

discard(*key*)

Discard element.

Parameters **key** – element to remove

pop(*last=True*)

Remove and return first or last element.

Parameters **last** – remove last element

Returns last element

mktempfile(*suffix=''*)

Create a temporary file with the given name suffix.

Parameters **suffix** (*str*) – extension of the file

Returns name of the temporary file

load_module(*filename*)

Load a Python module from a filename or qualified module name.

If filename ends with `.py`, the module is loaded from the given file. Otherwise it is taken to be a module name reachable from the path.

Example:

Parameters **filename** (*str*) – location of the module

Returns loaded module

Return type module

format_value(*data*, *precision=8*)
Pretty print a given value.

Parameters

- **data** – data to format
- **precision** (*int*) – max. number of digits

Returns pretty printed result

Return type str

format_tuple(*data*, *precision=8*, *columnsep='\t'*)
Pretty print a given tuple (or single value).

Parameters

- **data** – data to format
- **precision** (*int*) – max. number of digits
- **columnsep** (*str*) – column separator

Returns pretty printed result

Return type str

format_dictionary(*data*, *precision=8*, *keysep=':'*, *columnsep='\t'*)
Pretty print a given dictionary.

Parameters

- **data** (*dict*) – data to format
- **precision** (*int*) – max. number of digits
- **keysep** (*str*) – separator between key and value (default: `;`)
- **columnsep** (*str*) – column separator (default: `tab`)

Returns pretty printed result

Return type str

class UHeap(*key=None*)
Bases: `object`

Updatable heap.

Each element is represented as a pair (key, item). The operation `pop()` always returns the item with the smallest key. The operation `push(item)` either adds item (returns True) or updates its key (return False) A function for computing an item's key can be passed.

Parameters **key** – function for computing the sort key of an item

push(*item*)
Add the item or update it's key in case it already exists.

Parameters **item** – item to add

Returns True is item was not in the collection

pop()
Removes and returns the element with the smallest key.

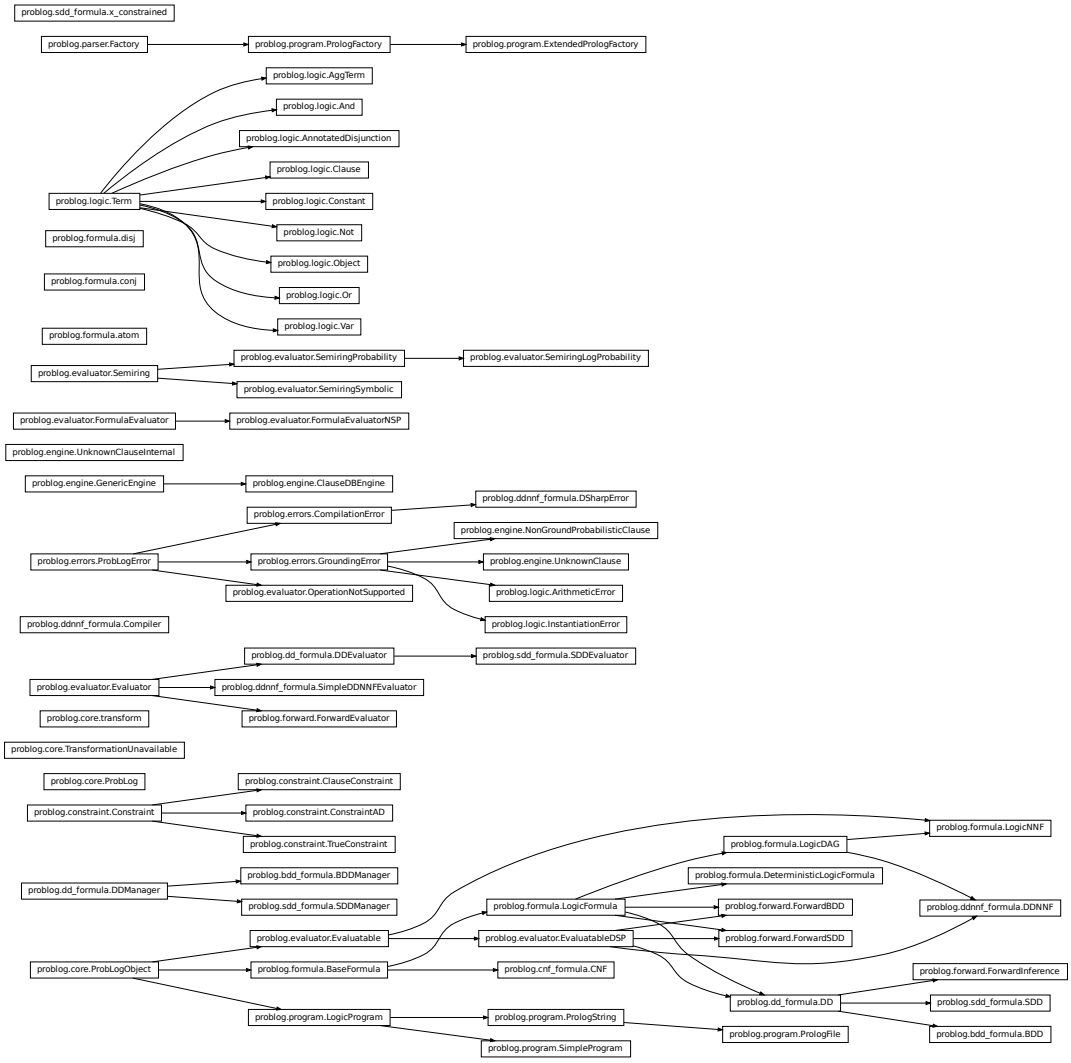
Returns item with the smallest key

pop_with_key()
Removes and returns the smallest element and its key.

Returns smallest element (key, element)

peek()
Returns the element with the smallest key without removing it.

Returns item with the smallest key



CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

A

B

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z