# SWI-Prolog `cplint` Pack Manual

Fabrizio Riguzzi
fabrizio.riguzzi@unife.it

December 20, 2015

## 1 Introduction

`cplint` is a suite of programs for reasoning with LPADs/CP-logic programs [10, 11, 8, 9]. It contains modules for both inference and learning.

## 2 Installation

`cplint` is distributed as a pack of SWI-Prolog. To install it, use

```
?- pack_install(cplint).
```

Moreover, in order to make sure you have a foreign library that matches your architecture, run

```
?-  pack_rebuild(cplint).
```

## 3 Syntax

LPAD and CP-logic programs consist of a set of annotated disjunctive clauses. Disjunction in the head is represented with a semicolon and atoms in the head are separated from probabilities by a colon. For the rest, the usual syntax of Prolog is used. For example, the CP-logic clause

$$h_1 : p_1 \vee \ldots \vee h_n : p_n \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_l$$

is represented by

```
h1:p1 ; ... ; hn:pn :- b1,...,bm,\+ c1,....,\+ cl
```

No parentheses are necessary. The `pi` are numeric expressions. It is up to the user to ensure that the numeric expressions are legal, i.e. that they sum up to less than one.

If the clause has an empty body, it can be represented like this

```
h1:p1 ; ... ; hn:pn.
```

If the clause has a single head with probability 1, the annotation can be omitted and the clause takes the form of a normal prolog clause, i.e.

```
h1 :- b1,...,bm,\+ c1,...,\+ cl.
```

stands for

```
h1:1 :- b1,...,bm,\+ c1,...,\+ cl.
```

The coin example of [11] is represented as (file `coin.cpl`)

```
heads(Coin):1/2 ; tails(Coin):1/2 :-
  toss(Coin),\+biased(Coin).

heads(Coin):0.6 ; tails(Coin):0.4 :-
  toss(Coin),biased(Coin).

fair(Coin):0.9 ; biased(Coin):0.1.

toss(coin).
```

The first clause states that if we toss a coin that is not biased it has equal probability of landing heads and tails. The second states that if the coin is biased it has a slightly higher probability of landing heads. The third states that the coin is fair with probability 0.9 and biased with probability 0.1 and the last clause states that we toss a coin with certainty.

Moreover, the bodies of rules may contain the built-in predicates:

```
is/2, >/2, </2, >=/2 ,=</2,
=:=/2, =\=/2, true/0, false/0,
=/2, ==/2, \=/2 ,\==/2, length/2
```

The bodies may also contain the following library predicates:

```
member/2, max_list/2, min_list/2
nth0/3, nth/3, dif/2, select/3
```

plus the predicate

```
average/2
```

that, given a list of numbers, computes its arithmetic mean.

# 4 Inference

`cplint` answers queries using the module `pita`. It performs the program transformation technique of [7]. Differently from that work, techniques alternative to tabling and answer subsumption are used.

## 4.1 Commands

The LPAD or CP-logic program must be stored in a text file with extension `.lpad` or `.cpl`. Suppose you have stored the example above in file `coin.cpl`. In order to answer queries to this program, you have to run SWI-Prolog and load `pita` by issuing the command

```
?- use_module(library(pita)).
```

at the command prompt. Then you must load the source file `coin.cpl` with the `pita` command

```
?- load(coin).
```

if `coin.cpl` is in the current directory, or

```
?- load('path_to_coin/coin').
```

if `coin.cpl` is in a different directory. `load(file)` loads `file.lpad` if it exists, or `file.cpl`s if it exists, or fails. `load_file(file)` loads `file` without adding the extension. At this point you can pose query to the program by using the `pita` predicate `prob/2` that takes as its first argument an atomic goal and returns the computed probability in its second argument. For example, the probability of `heads(coin)` can be asked with the query

```
?- prob(heads(coin),P).
```

After having loaded a program, future loadings add clauses. If you want to replace the current clauses with the new ones you have to restart SWI-Prolog.

Instead of preparing a file containing the LPAD/CP-logic program, you can prepare a Prolog file where you first load `pita` and then enclose the probabilistic clauses in `:-cplint.` and `:-end_cplint.` For example, the coin program above can be stored in `coin.pl` as follows

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

:-cplint.

heads(Coin):1/2 ; tails(Coin):1/2:-
toss(Coin),\+biased(Coin).

heads(Coin):0.6 ; tails(Coin):0.4:-
toss(Coin),biased(Coin).

fair(Coin):0.9 ; biased(Coin):0.1.

toss(coin).

:-end_cplint.
```

Then you can simply load `coin.pl` as

```
?-[coin].
```

and query it with

```
?- prob(heads(coin),P).
```

Note that supplying `coin.pl` as an argument to the command `swipl` currently returns errors due to bad interaction between `pita.pl` and the top-level. The program is loaded correctly anyway but it is recommended to load programs from the top-level to avoid these errors.

#### 4.1.1 Parameters

The module make use of a number of parameters in order to control its behavior. They that can be set with the command

```
?- set(parameter,value).
```

from the top-level after having loaded the module. The current value can be read with

```
pita_setting(parameter,Value).
```

from the top-level. The available parameters are:

- `epsilon_parsing`: if (1 - the sum of the probabilities of all the head atoms) is smaller than `epsilon_parsing`, then `pita` adds the null event to the head. Default value `0.00001`.
- `single_var`: determines how non ground clauses are treated: if `true`, a single random variable is assigned to the whole non ground clause, if `false`, a different random variable is assigned to every grounding of the clause. Default value `false`.
- `depth_bound`: if `true`, the depth of the derivation of the goal is limited to the value of the `depth` parameter. Default value `false`.
- `depth`: maximum depth of derivations when `depth_bound` is set to `true`. Default value `2`.

### 4.2 Files

The `packs/cplint/prolog/examples` folder in SWI-Prolog home contains some example programs. The `packs/cplint/doc` folder in SWI-Prolog home contains this manual in latex, html and pdf.

## 5 Learning

`cplint` contains the following learning algorithms:

- EMBLEM (EM over Bdds for probabilistic Logic programs Efficient Mining): an implementation of EM for learning parameters that computes expectations directly on BDDs [3, 1, 2]
- SLIPCOVER (Structure LearnIng of Probabilistic logic programs by searCh-ing OVER the clause space): an algorithm for learning the structure of programs by searching the clause space and the theory space separatery [4]

### 5.1 Input

To execute the learning algorithms, prepare four files in the same folder:

- `<stem>.kb`: contains the example interpretations

- `<stem>.bg`: contains the background knowledge, i.e., knowledge valid for all interpretations
- `<stem>.l`: contains language bias information
- `<stem>.cpl`: contains the LPAD for you which you want to learn the parameters or the initial LPAD for SLIPCASE and LEMUR. For SLIPCOVER, this file should be absent

where `<stem>` is your dataset name. Examples of these files can be found in the dataset pages.

In `<stem>.kb` the example interpretations have to be given as a list of Prolog facts initiated by `begin(model(<name>)).` and terminated by `end(model(<name>)).` as in

```
begin(model(b1)).
sameperson(1,2).
movie(f1,1).
movie(f1,2).
workedunder(1,w1).
workedunder(2,w1).
gender(1,female).
gender(2,female).
actor(1).
actor(2).
end(model(b1)).
```

The interpretations may contain a fact of the form

```
prob(0.3).
```

assigning a probability (0.3 in this case) to the interpretations. If this is omitted, the probability of each interpretation is considered equal to $1/n$ where $n$ is the total number of interpretations. `prob/1` can be used to set different multiplicity for the different interpretations.

`<stem>.bg` can contain Prolog clauses that can be used to derive additional conclusions from the atoms in the interpretations.

`<stem>.l` contains the declarations of the input and output predicates, of the unseen predicates and the commands for setting the algorithms' parameters. Output predicates are declared as

```
output(<predicate>/<arity>).
```

and define the predicates whose atoms in the input interpretations are used as the goals for the prediction of which you want to optimize the parameters. Derivations for these goals are built by the systems.

Input predicates are those for the predictions of which you do not want to optimize the parameters. You can declare closed world input predicates with

```
input_cw(<predicate>/<arity>).
```

For these predicates, the only true atoms are those in the interpretations, the clauses in the input program are not used to derive atoms not present in the interpretations.

Open world input predicates are declared with

`input(<predicate>/<arity>).`

In this case, if a subgoal for such a predicate is encountered when deriving the atoms for the output predicates, both the facts in the interpretations and the clauses of the input program are used.

For SLIPCOVER, you have to specify the language bias by means of mode declarations in the style of Progol.

`modeh(<recall>,<predicate>(<arg1>,...).`

specifies the atoms that can appear in the head of clauses, while

`modeb(<recall>,<predicate>(<arg1>,...).`

specifies the atoms that can appear in the body of clauses. `<recall>` can be an integer or `*` (currently unused).

The arguments are of the form

`+<type>`

for specifying an input variable of type `<type>`, or

`-<type>`

for specifying an output variable of type `<type>`. or

`<constant>`

for specifying a constant.

SLIPCOVER also allows the arguments

`#<type>`

for specifying an argument which should be replaced by a constant of type `<type>` in the bottom clause but should not be used for replacing input variables of the following literals or

`-#<type>`

for specifying an argument which should be replaced by a constant of type `<type>` in the bottom clause and that should be used for replacing input variables of the following literals. `#` and `-#` differ only in the creation of the bottom clause.

An example of language bias for the UWCSE domain is

```
output(advisedby/2).

input(student/1).
input(professor/1).
....

modeh(*,advisedby(+person,+person)).

modeb(*,professor(+person)).
modeb(*,student(+person)).
modeb(*,sameperson(+person, -person)).
```

```
modeb(*,sameperson(-person, +person)).
modeb(*,samecourse(+course, -course)).
modeb(*,samecourse(-course, +course)).
....
```

SLIPCOVER also requires facts for the `determination/2` predicate that indicate which predicates can appear in the body of clauses. For example

```
determination(professor/1,student/1).
determination(student/1,hasposition/2).
```

state that `student/1` can appear in the body of clauses for `professor/1` and that `hasposition/2` can appear in the body of clauses for `student/1`.

SLIPCOVER also allows mode declarations of the form

```
modeh(<r>,[<s1>,...,<sn>],[<a1>,...,<an>],[<P1/Ar1>,...,<Pk/Ark>]).
```

These mode declarations are used to generate clauses with more than two head atoms. In them, `<s1>,...,<sn>` are schemas, `<a1>,...,<an>` are atoms such that `<ai>` is obtained from `<si>` by replacing placemarkers with variables, `<Pi/Ari>` are the predicates admitted in the body. `<a1>,...,<an>` are used to indicate which variables should be shared by the atoms in the head. An example of such a mode declaration is

```
modeh(*,
  [advisedby(+person,+person),tempadvisedby(+person,+person)],
  [advisedby(A,B),tempadvisedby(A,B)],
  [professor/1,student/1,hasposition/2,inphase/2,
  publication/2,taughtby/3,ta/3,courselevel/2,yearsinprogram/2]).
```

## 5.2 Parameters

In order to set the algorithms' parameters, you have to insert in `<stem>.l` commands of the form

```
:- set(<parameter>,<value>).
```

The available parameters are:

- `depth` (values: integer or `inf`, default value: 3, , valid for EMBLEM and SLIPCOVER): depth of derivations if `depth_bound` is set to `true`
- `single_var` (values: `{true,false}`, default value: `false`, valid for EMBLEM and SLIPCOVER): if set to `true`, there is a random variable for each clause, instead of a different random variable for each grounding of each clause
- `sample_size` (values: integer, default value: 1000, , valid for EMBLEM and SLIPCOVER): total number of examples when the models in the `.kb` file contain a `prob(P).` fact. In that case, one model corresponds to `sample_size*P` examples
- `epsilon_em` (values: real, default value: 0.1, valid for EMBLEM and SLIPCOVER): if the difference in the log likelihood in two successive EM iteration is smaller than `epsilon_em`, then EM stops

- `epsilon_em_fraction` (values: real, default value: 0.01, valid for EMBLEM and SLIPCOVER): if the difference in the log likelihood in two successive EM iteration is smaller than `epsilon_em_fraction`*(-current log likelihood), then EM stops
- `iter` (values: integer, defualt value: 1, valid for EMBLEM and SLIPCOVER): maximum number of iteration of EM parameter learning. If set to -1, no maximum number of iterations is imposed
- `iterREF` (values: integer, defualt value: 1, valid for SLIPCOVER): maximum number of iteration of EM parameter learning for refinements. If set to -1, no maximum number of iterations is imposed.
- `random_restarts_number` (values: integer, default value: 1, valid for EMBLEM and SLIPCOVER): number of random restarts of EM learning
- `random_restarts_REFnumber` (values: integer, default value: 1, valid for SLIPCOVER): number of random restarts of EM learning for refinements
- `setrand` (values: rand(integer,integer,integer), valid for EMBLEM and SLIPCOVER): seed for the random functions, see Yap manual for allowed values
- `logzero` (values: negative real, default value $\log(0.000001)$, valid for SLIPCOVER): value assigned to $\log 0$
- `max_iter` (values: integer, default value: 10, valid for SLIPCASE and SLIPCOVER): number of interations of beam search
- `max_var` (values: integer, default value: 1, valid for SLIPCOVER): maximum number of distinct variables in a clause
- `verbosity` (values: integer in [1,3], default value: 1): level of verbosity of the algorithms
- `beamsize` (values: integer, default value: 20, valid for SLIPCOVER): size of the beam
- `megaex_bottom` (values: integer, default value: 1, valid for SLIPCOVER): number of mega-examples on which to build the bottom clauses
- `initial_clauses_per_megaex` (values: integer, default value: 1, valid for SLIPCOVER): number of bottom clauses to build for each mega-example
- `d` (values: integer, default value: 10000, valid for SLIPCOVER): number of saturation steps when building the bottom clause
- `max_iter_structure` (values: integer, default value: 1, valid for SLIPCOVER): maximum number of theory search iterations
- `background_clauses` (values: integer, default value: 50, valid for SLIPCOVER): maximum numbers of background clauses
- `maxdepth_var` (values: integer, default value: 2, valid for SLIPCOVER): maximum depth of variables in clauses (as defined in [5]).

- `score` (values: `ll`, `aucpr`, default value `ll`, valid for SLIPCOVER): determines the score function for refinement: if set to `ll`, log likelihood is used, if set to `aucpr`, the area under the Precision-Recall curve is used.

## 5.3 Commands

To execute EMBLEM, load `slipcover` with

```
?- use_module(library(slipcover)).
```

and call

```
?- em(stem).
```

To execute SLIPCOVER, load `slipcover` with

```
?- use_module(library(slipcover)).
```

and call

```
?- sl(stem).
```

## 5.4 Testing

To test the theories learned, load `test.pl` with

```
?- use_module(library(test)).
```

and call

```
?- main([<stem_fold1>,...,<stem_foldn>],[<testing_set_fold1>,...,
  <testing_set_foldn>]).
```

For example, if you want to test the theory in `ai_train.rules` on the set `ai.kb`, you can call

```
?- main([ai_train],[ai]).
```

The testing program has the following parameters:

- `neg_ex` (values: `given`, `cw`, default value: `cw`): if set to `given`, the negative examples are taken from `<testing_set_foldi>.kb`, i.e., those example `ex` stored as `neg(ex)`; if set to `cw`, the negative examples are generated according to the closed world assumption, i.e., all atoms for target predicates that are not positive examples. The set of all atoms is obtained by collecting the set of constants for each type of the arguments of the target predicate.

The testing program produces the following output in the current folder:

- `cll.pl`: for each fold, the list of examples orderd by their probability of being true
- `areas.csv`: the areas under the Precision-Recall curve and the Receiver Operating Characteristic curve
- `curve_roc.m`: a Matlab file for plotting the Receiver Operating Characteristic curve
- `curve_pr.m`: a Matlab file for plotting the Precision-Recall curve

### 5.5 Learning Examples

The `cplint/prolog/examples` folder of SWI-Prolog home contain examples of input and output files for the learning algorithms

- `ai_train_par`: parameter learning example from UWCSE
- `ai_train`: structure learning example from UWCSE

## 6 License

`cplint` follows the Artistic License 2.0 that you can find in `cplint` root folder. The copyright is by Fabrizio Riguzzi.

The library CUDD for manipulating BDDs has the following license:



## References

[1] Elena Bellodi and Fabrizio Riguzzi. EM over binary decision diagrams for probabilistic logic programs. In *Proceedings of the 26th Italian Conference on Computational Logic (CILC2011), Pescara, Italy, 31 August 31-2 September, 2011*, 2011.

[2] Elena Bellodi and Fabrizio Riguzzi. EM over binary decision diagrams for probabilistic logic programs. Technical Report CS-2011-01, Dipartimento di Ingegneria, Università di Ferrara, Italy, 2011.

[3] Elena Bellodi and Fabrizio Riguzzi. Expectation Maximization over binary decision diagrams for probabilistic logic programs. *Intel. Data Anal.*, 16(6), 2012.

[4] Elena Bellodi and Fabrizio Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming*, 2013.

[5] William W. Cohen. Pac-learning non-recursive prolog clauses. *Artif. Intell.*, 79(1):1–38, 1995.

[6] G. Elidan and N. Friedman. Learning hidden variable networks: The information bottleneck approach. *Journal of Machine Learning Research*, 6:81–127, 2005.

[7] Fabrizio Riguzzi and Terrance Swift. Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In *Technical Communications of the International Conference on Logic Programming*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 162–171. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[8] J. Vennekens, M. Denecker, and M. Bruynooghe. Representing causal information about a probabilistic process. In *Proceedings of the 10th European Conference on Logics in Artificial Intelligence*, LNAI. Springer, September 2006.

[9] J. Vennekens, Marc Denecker, and Maurice Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory Pract. Log. Program.*, 9(3):245–308, 2009.

[10] J. Vennekens and S. Verbaeten. Logic programs with annotated disjunctions. Technical Report CW386, K. U. Leuven, 2003.

[11] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, volume 3131 of *LNCS*, pages 195–209. Springer, 2004.