Library PCC(FD) documentation

Matthieu Petit[⋆] and Arnaud Gotlieb

IRISA-INRIA, Campus de Beaulieu, 35042 Rennes cedex, France
{Matthieu.Petit,Arnaud.Gotlieb}@irisa.fr

Abstract. This documentation describes the PCC(FD) library v.0. that can be downloaded from http://www.irisa.fr/lande/petit/index.en.html. The library has been built upon SICStus 3.11.0 [2].

1 Usage

To use the library PCC(FD),the file `ctr_proba_v0.pl` should be loaded. The library adds three probabilistic constraint combinators to the clp(FD) library of SICStus Prolog [1].

```
?- [ctr_proba_v0.pl].
```

For some technical reasons, each combinator uses a global variable `Env`. This variable must be intialized by using `init_env/1`. This variable contains different common parameters to probabilistic contraint combinators. For example, the following example simulates a dice drawing.

```
?- init_env(Env),choose(X,[1,2,3,4,5,6]-[1,1,1,1,1,1],[X=Dice],[],Env).
```

2 Probabilistic terminal configuration predicate

The predicate `ptc(Goal,Var_List,Result)` computes empirically the set of terminal configurations in PCC(FD) [3]. Given a **list** of Prolog goals `Goal` along with a list of variables `Var_List`, the `ptc` predicates launches a given number of `Goal` runs, records the resulting constraint store projection (i.e. projection of domains on `Var_List`) after the constraint propagation step, and computes the occurrence rate of each constraint store projection. By using this predicate, one can study the probabilistic behaviour of our constraint combinators in PCC(FD). The number of iterations are fixed at 5000 but can be modified by redifining the predicate `nb_iteration/1`.

3 Combinators

The library PCC(FD) is composed by three combinators:

- `choose`, where the domain and probability distribution of the probabilistic choice are a list of finite domain variables;
- `choose_range`, where the domain of the probabilistic choice is a range represented with two distinct FD variables Min and Max, and its probability distribution a list of finite domain variables;
- `choose_decision`, where the domain of the probabilistic choice is the boolean domain $\{0, 1\}$ and its probability distribution is a couple of distinct finite domain variables.

Note that `choose_range` and `choose_decision` can be rewritten by using `choose`. However, a dedicated filtering algorithm has been implemented for a more efficient behaviour of combinators.

[⋆] This work is part of the GENETTA project granted by the Brittany region: http://www.irisa.fr/lande/petit/genetta.

3.1 `choose` constraint combinator

The probabilistic constraint combinator `choose` models a random choice of a value for a random variable X given a domain and a probability distribution.

```
choose(X,[V1,..Vn]-[W1,..,Wn],Goal,Options,Env)
```

where X is a random variable, `V1,..,Vn,W1,..,Wn` are finite domain variables, `Goal` is a list of Prolog goal. `Options` is used to parameterize the filtering capacities of the combinator into the constraint propagation mechanism.

- `domain_unbound` must be used when variables of `[V1,..Vn]` are instantiated;
- `inconsistency_check` can be used to check the consistency of all the possible values V of `X` with respect to `Goal`. This option launchs a fix point computation which tries iteratively to prune the domain of `X,V1,..Vn,W1, ..,Wn`. The fix point computation is based on a inconsistency test between the possible values of `X` and `Goal`. This option is useful to improve the deduction of the combinator but are more costly in computation time;
- `lvar(L)` can be used to enrich the list of variables on which the combinator is awaked. This option is useful to parameterize the awaking conditions of the combinator;
- `no_filtering` can be employed to switch off the pruning capacities of the filtering algorithm. This option is useful to estimate the effectiveness of the filtering algorithm;
- `rv(U)` can be used to obtain the value of the random variable which is used to simulate the `X` value. This options is useful to verify the corectness of the probabilistic constraint combinator.

3.2 `choose_range` constraint combinator

The `choose_range` combinator implements a probabilistic choice operator for a range of values. The range is given by `[Xmin,Xmax]`, where `Xmin` and `Xmax` are two finite domain variables. `Xmin` denotes the lower bound of X wheras `Xmax` denotes its upper bound. The syntax of the combinator `choose_range` is as follows:

```
choose_range(X,[Xmin,Xmax]-Distribution,Goal,Options,Env)
```

where X is a random variable, `Xmin,Xmax` are finite domain variables, `Distribution` is a atom `'uniform'` which define a uniform probability distribution or a list of finite domain variables and `Goal` is a list of Prolog goal. `inconsistency_check`, `lvar(L)`, `no_filtering` and `rv(U)` options are available.

3.3 `choose_decision` constraint combinator

The `choose_decision` combinator implements a probabilistic boolean choice between two processes. This probabilistic boolean choice is represented as a list `[W1,W2]` of two finite domain variables. The term `neg(Constraint)` denotes the negation of `Constraint`. Note that `neg` is limited to simple arithmetic constraints composed of `#=`, `#\=`, `#>`, `#>=`, `#<`, `#=<`, `#/\` and `#\/`.

```
choose_decision(Constraint,[W1,W2],Goal1,Goal2,Options,Env)
```

where `X` is a random variable, `W1,W2` are two finite domain variables, `Goal1` and `Goal2` are a list of Prolog goal. `inconsistency_check`, `lvar(L)`, `no_filtering` and `rv(U)` options are available.

4 Examples

Three examples of the combinator usage is presented in this section

4.1 Dice playing

The following example extracted from [5] illustrates the use of `choose`. The `dice/1` goal modelizes a biased dice drawing. The bias of the dice is partially unknown. Bias knowledge is represented by constraints on the variables of the probability distribution. In the example, the probability to draw 6 is two times bigger than the probability to draw 1.

```
dice(Dice) :-
    init_env(Env),
    P1 in 1..4,
    P2 #= 2,
    P3 #= 2,
    P4 #= 2,
    P5 #= 2,
    P6 in 1..4,
    2*P1 #= P6,
    choose(X,[1,2,3,4,5,6]-[P1,P2,P3,P4,P5,P6],[Dice=X],[],Env).

? - ptc([dice(Dice)],[Dice],Result).

Result=[(Dice=1,0.07735),(Dice in 1..2,0.09065),
        (Dice=2,0.06285),(Dice in 2..3,0.10175),
        (Dice=3,0.05135),(Dice in 3..4,0.11795),
        (Dice=4,0.03860),(Dice in 4..5,0.12775),
        (Dice=5,0.02575),(Dice in 5..6,0.1401),
        (Dice=6,0.16590)]
```

The results show the different constraint store projections on X obtained after the constraint propagation step. For example, `(Dice=1,0.07735)` means that `Dice` is equal to 1 with a probability `0.07735`.

4.2 Primal Testing

The following example extracted from [5] illustrates the use of `choose_range`. The goal modelizes a weakest version of the Miller-Rabin primal testing.

```
primal_testing(N,K) :-
        init_env(Env),
        N in 3..4000000,
        Xmax#=N-1,
        itere(N,Xmax,K,Env).


itere(_N,_Xmax,0,_Env) :-
        !.

itere(N,Xmax,K,Env) :-
        choose_range(X,[2,Xmax],'uniform',[fermat_test(X,N)],[lvar([N])],Env),
        K1 is K-1,
        itere(N,Xmax,K1,Env).
```

4.3 Structural statiscal testing

The following example extracted from [5] illustrates the use of `choose_decision`. The goal modelizes the transformation of a problem structural statiscal testing [6] for the **foo** program (FIG. 1) into a probabilistic constraint problem [4].

```
int foo(int x, int y) {
1.  if (x =< 100  &&  y =< 100)
    {
2.      if (y > x + 50)
3.          ...
4.      if (x * y < 100)
5.          ...
6.  }
```

Fig. 1. Program foo

```
foo(X,Y, [W1,W2,W3,W4,W5,W6]) :-
    init_env(Env),
    X in 0..1000,Y in 0..1000,
    choose_decision(X#=<100#/\Y#=<100,[W1,W2],
    [choose_decision(Y#>X+50,[W3,W4],[],[],Env),
      choose_decision(Y*X#<100,[W5,W6],[],[],Env)],[],[],Env),
    N1*W1#=N2*W2,
    N3*W3#=N4*W4,
    N5*W5#=N6*W6.
```

5 Referencing this library

When referring to this implementation, please use `http://www.irisa.fr/lande/petit/tools.html`. The constraint combinators behaviour and the filtering algorithm associated to it is more precisely described in [5].

References

1. M. Carlsson, G. Ottosson, and B. Carlson. An Open–Ended Finite Domain Constraint Solver. In *Proc. of Programming Languages: Implementations, Logics, and Programs*, 1997.
2. Mats Carlsson. *SICStus Prolog User's Manual*. Swedish Institute in Computer Science, 1997.
3. P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). Technical Report CS-93-02, Brown University, 1993.
4. M. Petit and A. Gotlieb. An ongoing work on statistical structural testing via probabilistic concurrent constraint programming. In *Proc. of SIVOES-MODEVA workshop*, St Malo, France, November 2004.
5. M. Petit and A. Gotlieb. Probabilistic choice operators as constraints. In *INRIA Reserch Report. To appear*, 2006.
6. P. Thévenod-Fosse and H. Waeselynck. An investigation of statistical software testing. *Journal of Sotware Testing, Verification and Reliability*, 1(2):5–25, July 1991.