Piloting PyCSP^3 Solvers with General Options

Christophe Lecoutre and Charles Prud'homme

CRIL & LINA

February 4, 2022

Abstract. This document lists the general options that can be used to pilot embedded solvers, which are directly run when compiling PyCSP^3 models. Currently, ACE [6] and Choco [10] are the two first embedded solvers. Additional solvers are expected to be embedded in the near future.

1 Introduction

For generating an XCSP^3 file from a PyCSP^3 model (see pycsp.org for all details about the Python library PyCSP^3), you have to execute the following command:

```
python <file> [options]
```

where:

- `<file>` is a Python file that contains a PyCSP^3 model
- `[options]` are possible options that can be used when compiling

Among the options[1], we find:

- `-solve` that attempts to solve the instance with the embedded solver ACE while using default values for the options of the solver. It requires that Java version 8 (at least) is installed.
- `-solver=<solver_name>` that attempts to solve the instance with the solver whose name is given. Currently, it can be 'ace' or 'choco'. It requires that Java version 8 (at least) is installed.
- `-solver=[<solver_name>,<solver_options>]` that attempts to solve the instance with the solver whose name is given, while following the specified *general* solver options. The general options are assumed to be valid whatever is the selected solver. Note that we use square brackets (i.e., the symbols '[' and ']') to specify a list of terms with the symbol ',' used as a separator (and no tolerated whitespace). The options are then given in sequence with ',' acting as separator. If an option is complex (i.e., needs more than a single piece of information), the square brackets are recursively used to specify them.

[1] Other options concerning data and model are described in: http://pycsp.org/documentation/interface/CommandInterface/

Among the *general* solver options (used when one wants to directly solve a problem instance while compiling a PyCSP^3 model), specified to pilot the solver when using `-solver`, we find:

- a limit on search with `limit`
- a variable ordering heuristic with `varHeuristic`
- a value ordering heuristic with `valHeuristic`
- a complementary technique for guiding search with `lc`, `cos` and `last`
- a restart policy with `restarts`
- objective lower and upper bounds (for optimization) with `lb` and `ub`
- a seed (for a random process) with `seed`
- a verbose mode with `v` (and also `vv` or `vvv`)
- a trace of the search process with `trace`

and we can also use:

- some *specific* arguments for a given solver with `args`

For example,

```
python3 Zebra.py -solver=[choco,limit=60s,varh=dom/wdeg,lc,v]
```

compiles the model *Zebra.py* and solves it with Choco while limiting search to at most 60 seconds, using the classical variable ordering heuristic *dom/wdeg* with last-conflict resoning activated, and displaying information in verbose mode.

If ever you want to do the same thing with ACE, you just replace the name of the solver in the command line:

```
python3 Zebra.py -solver=[ace,limit=60s,varh=dom/wdeg,lc,v]
```

2 Solver Options

In this section, we provide some details about the (general) solver options.

2.1 Limit

To set a limit on the solver, you must use the option `limit` followed by the symbol '=', an integer, and finally a limit unit that can be:

- `h` for a number of hours
- `m` for a number of seconds
- `s` for a number of seconds
- `sols` for a number of solutions (relevant for decision problems)
- `runs` for a number of runs (relevant if a restart policy is used)

You can also combine several limits by putting them between square brackets. The solver stops as soon as a limit is reached. For example,

```
    limit=[20m,50runs]
```

indicates that the solver must stop when it has been running for 20 minutes or when 50 runs have been executed.

When solving a CSP instance, the default behaviour of the embedded solvers is to determine whether a solution exists or not, exhibiting the first found solution when it exists. However, in some cases, one may want to compute all solutions. This is possible with `limit=no`.

For example,

```
python Zebra.py -solver=[ace,limit=no]
```

computes all solutions for the model/problem *Zebra.py* with the embedded solver ACE.

2.2 Variable Ordering Heuristic

For telling the solver to adopt a specific variable ordering heuristic, you must use the option `varHeuristic`, or equivalently `varh`, followed by the symbol '=' and a name among:

- `input` to select variables according to their order in the input file; this is sometimes called `lexico` in the literature
- `dom` to select variables according to the size of the current domains [5]
- `rand` to select variables randomly
- `ibs` to use impact-based search [11]; as a consequence, the option `valHeuristic` must not been used since a pair (variable,value) is actually selected
- `abs` to use activity-based search [9]; as a consequence, the option `valHeuristic` must not been used since a pair (variable,value) is actually selected
- `impact` to select variables according to ibs (but values can be selected by any value ordering heuristic)
- `activity` to select vaariables according to abs (but values can be selected by any value ordering heuristic)
- `dom/ddeg` to select variables according to the ratio 'current domain size' to 'dynamic degree' [1]
- `dom/wdeg` to select variables according to constraint weighting [2]

2.3 Value Ordering Heuristic

For telling the solver to adopt a specific value heuristic, you must use the option `valHeuristic`, or equivalently `valh`, followed by the symbol '=' and a name among:

- `min` to select the minimal value in the current domain of the selected variable
- `max` to select the maximal value in the current domain of the selected variable
- `med` to select the median value in the current domain of the selected variable
- `mid` to select the value in the middle of the domain
- `rand` to select values randomly
- `best` to select the best value according to BIVS (relevant for COP) [3]

2.4 Techniques to Go with Heuristics

For telling the solver to use last-conflict reasoning [7] , you must use the option `lastConflict`, or equivalently `lc`, followed by the symbol '=' and an integer. Note that `lc` alone is accepted, and is equivalent to `lc=1`.

For using conflict ordering search [4], you must use the option `cos`.

For using progress (or phase) saving, i.e., the fact of selecting in priority for a variable the value assigned to it in the last solution, if a solution has already been found and if the value is still present in the current domain, you must use the option `last`.

2.5 Restarts

To use a restart policy, you must use the option `restarts` followed by the symbol '=' and a name among:

- `monotonic`
- `geometric`
- `luby`

It is also possible to use other arguments for `restarts` (then, arguments are put beteween square brackets after the symbol '='). For all three policies, it is possible to set the value of the initial cutoff, i.e., the one used to stop the first run. For `geometric`, it is also possible to indicate the factor used to increase the value of the cutoff between two runs. You must use:

- `cutoff` followed by '=' and an integer
- `factor` followed by '=' and an integer

If κ denotes the value of the initial cutoff, and ϕ the geometric factor, the length (cutoff) of the ith run is:

- $\kappa * i$ for the monotonic restart policy
- $l(i-1)\phi$ for the geometric restart policy where $l(i-1)$ is the length of the i-1th run, and $l(1) = \kappa$.

For example, to express a geometric restart policy, of initial cutoff 100 and factor 1.2, we write:

```
restarts=[geometric,cutoff=100,factor=1.2]
```

Note that the default value is expected to be 10.

2.6 Lower and Upper Bounds

When solving a COP instance, one may wish to indicate a lower bound and/or an upper bound concerning the objective value. You can use the options

- `lb` followed by '=' and an integer, indicating that the solver should not seek solutions of costs less than or equal to the specified value
- `ub`, followed by '=' and an integer, indicating that the solver should not seek solutions of costs greater than or equal to the specified value

2.7 Seed

When the solver is expected to use random numbers, it is possible to initialize the random generator with a specific seed. To set a seed, you must use the option `seed` followed by the symbol '=' and an integer. For example:

```
seed=100
```

2.8 Output in Verbose Mode

In addition to the normal mode (when the option is not used at all), you can choose among:

- `v` , or equivalently, `verbose`
- `vv` for very verbose
- `vvv` for very very verbose

Important When a verbose mode is used or when the option `-solve` is used, the output of the solver is shown normally. In normal mode, the output of the solver is intercepted, and a succinct information is given after the solving process.

The output follows a specific format (to be defined).

2.9 Trace

For displaying the trace of the solver, you must use the option `trace`: the trace is displayed in standard system output. It is also possible to indicate the name of a file after the symbol '=', as in:

```
trace=traceExample.txt
```

The trace follows a specific format (to be defined).

2.10 Specific Arguments

Some arguments remain specific to solvers. For example, when using ACE, you can indicate that you want to use the propagator (filtering algorithm) STR2 [8] for positive table constraints with `-positive=str2`. Then, you must use `args` with a string (between double quotes) as value. This string will be concatenated, exactly as it is written, after inserting a whitespace, to the command line used to run the solver.

For example,

```
python3 Crosswords.py -data=h0504.json -solver=[ace,args="-positive=str2"]
```

compiles the model *Crosswords*, with the data file *h0504.json*, and solves it with ACE while using the algorithm STR2 for the positive table constraints.

Acknowledgements

This work has been supported by the project CPER DATA from the Hauts-de-France Region.

References

1. C. Bessiere and J. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP'96*, pages 61–75, 1996.
2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
3. J.-G. Fages and C. Prud'homme. Making the first solution good! In *Proceedings of ICTAI'17*, pages 1073–1077, 2017.
4. S. Gay, R. Hartert, C. Lecoutre, and P. Schauss. Conflict ordering search for scheduling problems. In *Proceedings of CP'15*, pages 140–148, 2015.
5. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
6. C. Lecoutre. ACE (AbsCon Essence), v2.0, CRIL, CNRS, Univ. Artois. 2021. https://github.com/xcsp3team/ace.
7. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Reasonning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.
8. Christophe Lecoutre. STR2: Optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.
9. L. Michel and P. Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Proceedings of CPAIOR'12*, pages 228–243, 2012.
10. C. Prud'homme, J.-G. Fages, and X. Lorca. Choco-solver, TASC, INRIA Rennes, LINA, Cosling S.A. 2016. https://choco-solver.org/.
11. P. Refalo. Impact-based search strategies for constraint programming. In *Proceedings of CP'04*, pages 557–571, 2004.