First XCSP3 Competition
(CSP and COP competition)

Last Call for Solvers and Benchmarks

(June 1, 2017)

http://www.xcsp.org

The first international XCSP3 constraint solver competition is organized to improve our knowledge about components (e.g., filtering algorithms, heuristics, search strategies, and learning procedures) that are behind the efficiency of solving systems (referred to as constraint solvers in this document) for combinatorial constrained problems. Two classical problems are considered for this competition:

- CSP (Constraint Satisfaction Problem)
- COP (Constrained Optimization Problem)

The intermediate[1] format XCSP3 is used as input format for the solvers. The effort required for entering the competition is limited because some tools (parsers) are available, and only a central set of popular (and important) constraints is considered.

This call for solvers and benchmarks presents the tracks that will be considered during the competition. In particular, we give important details about the format restrictions, the execution environment, and the rules that must be followed by the solvers.

[1]XCSP3 is neither a modeling language, nor a flat format. It is intermediate because it preserves the structure of problems through the concepts of variable arrays, constraint groups/blocks, and meta-constraints.

Contents

1 Timetable

The deadlines of the competition are defined below:

Opening of the registration site at http://www.cril.fr/CompetitionXCSP17/	May 2017
Pre-registration of contestants	May 20, 2017
Final registration (submission of solvers and benchmarks)	June 10, 2017
Test of solvers conformance	mid-June 2017
Competition running	Summer 2017
Final results available	during CP 2017

Once submitted, solvers will be run on a limited number of benchmarks to make sure that they interact correctly with the evaluation environment. Potential problems will be reported to the authors by the 17th of June 2017. Bug fixes will be due by the 22th of June 2017.

2 Tracks

First, do note that we consider two main problems: CSP (a decision problem) and COP (an optimization problem). For CSP, the goal is to exhibit one solution or to prove that none exists. For COP (mono-objective optimization), the goal is to exhibit a solution with the best possible objective value, ideally proving that it represents an optimum solution.

Anyone can submit a solver to any particular track. There are exactly 8 tracks. The first 6 tracks impose absolutely no conditions on solvers. For example, they can be written in any language (provided that we can reasonably execute them in our environment), and can be complete or incomplete solvers (e.g., based on local search). The 6 Standard tracks are described by Table 1.

Problem	Goal	Exploration	Timeout
CSP	one solution	sequential	40 minutes
CSP	one solution	parallel	40 minutes
COP	best solution	sequential	4 minutes
COP	best solution	parallel	4 minutes
COP	best solution	sequential	40 minutes
COP	best solution	parallel	40 minutes

Table 1: Standard Tracks.

Ranking for COP will be stated in two different manners: by considering and not considering possible proofs of optimality, permitting in the latter case to emphasize the quality of incomplete solvers.

There are also 2 Mini-Solver tracks, where a mini-solver is a solver whose code must be open-source and composed of less than $8,000$ lines of at most 160 characters (while discarding comments, code for parsing XCSP3 and code of standard libraries). For Mini-solvers tracks, only sequential exploration is considered, and the set of constraints is rather restricted, as described in the next section.

Our interest in Mini-solver tracks is two-fold. First, this should facilitate the participation of people (e.g., students) whose solvers cannot be complete enough to compete with well-established solvers from the community. Second, it will hopefully provide the community with one (or several) compact well-documented and easily extendable open-source solver, especially as there is a special price delivered by the jury about clarity and reusability of mini-solvers.

Problem	Goal	Exploration	Timeout
CSP	one solution	sequential	40 minutes
COP	best solution	sequential	40 minutes

Table 2: Mini-Solver Tracks.

3 Format

The complete description of the format (XCSP3) used to represent combinatorial constrained problems can be found in [BLP16]. However, for the 2017 competition, we limit XCSP3 to its kernel, called XCSP3-core. This means that the scope of XCSP3 is restricted to:

- integer variables,
- CSP and COP problems,
- a set of 20 popular (global) constraints for Standard tracks, and generic constraints for Mini-solver tracks.

For simplicity, we also impose the following restrictions:

- Integer variables have finite domains (and so, the special value *infinity* is forbidden).
- Variable arrays always start indexing at 0 (and so, the attribute `startIndex`, whose default value is 0, cannot be associated with `<array>`).
- The attribute `as` can only be associated with elements `<var>` and `<array>`; see Section 10.5 in [BLP16].
- Undefined variables are not accepted but useless variables are (note that parsers/solvers can easily identify useless variables); see Section 2.10 in [BLP16].
- Advanced forms of constraints (see Part III in [BLP16]) are not accepted, except for the very specific cases explicitly described in the rest of this section.
- Reification and views are not accepted.
- The type of the objective (in case of a COP instance) cannot be `"product"` or `"lex"`, and when it is `"expression"`, the content of the element `<minimize>` or `<maximize>` can only be a variable (identifier); see Chapter 3 in [BLP16].
- Any integer value occurring in an XCSP3 file must belong to the interval $-2^{31}..2^{31}-1$.

3.1 Constraints for Standard Tracks

Twenty constraints are involved in Standard tracks.

One might think that 20 constraints is too much. However, it turns out that specific code of propagators is needed for approximately 13 constraints only, because:

- similar propagators may be used for `regular` and `mdd`,
- similar propagators may be used for `maximum` and `minimum`,
- `channel` can be decomposed into `element` constraints,
- `ordered`, `allEqual` and `instantiation` can be *trivially* reformulated as `intension`,
- `slide` is decomposed into a set of constraints `intension` or `extension`; the parser can do it automatically for you.

Maybe, you might think that 20 constraints is not much enough. Just note that a large majority of the 23,000 instances (from around 90 classical models, some of them being described for example at CSPLib) that are currently available on our website only involve these 20 constraints.

3.1.1 Constraint `intension`

This constraint is described in Section 4.1.1.1 in [BLP16]. There is no competition restriction for this constraint.

3.1.2 Constraint `extension`

This constraint is described in Section 4.1.1.2 in [BLP16]. Competition restrictions:

1. Short tables (i.e., tables with tuples containing '*') and compressed tables (i.e., tables with compressed tuples) are not accepted.
2. Empty Tables (i.e., tables with with 0 support or 0 conflict) are not accepted.

Note that unary, binary and n-ary extensional constraints are accepted.

3.1.3 Constraint `regular`

This constraint is described in Section 4.1.2.1 in [BLP16]. Competition restrictions:

1. The automaton on which is based the constraint must be deterministic.

3.1.4 Constraint `mdd`

This constraint is described in Section 4.1.2.3 in [BLP16]. Competition restrictions:

1. There must be at least one path from the root node to the terminal node.
2. In `<transitions>`, the root note is given by the first item of the first transition, and the terminal node is given by the third item of the last transition.

3.1.5 Constraint `allDifferent`

This constraint is described in Section 4.1.3.1 in [BLP16]. Competition restrictions:

1. If present, the element `<except>` only contains one (integer) value.
2. Restricted forms (obtained by using the attribute `restriction`) are not accepted.

In addition to the basic form of `allDifferent`, the advanced form `allDifferent-matrix` described in Section 7.2.1 in [BLP16] is accepted.

3.1.6 Constraint `allEqual`

This constraint is described in Section 4.1.3.2 in [BLP16]. There is no competition restriction for this constraint.

3.1.7 Constraint `ordered`

This constraint is described in Section 4.1.3.4 in [BLP16]. Competition restrictions:

1. The compact form, obtained by using the attribute `case`, is not accepted.

3.1.8 Constraint `lex`

This constraint is described in Section 7.1.4.1 in [BLP16]. There is no competition restriction for this constraint.

In addition to this form of `lex`, the advanced form `lex-matrix` described in Section 7.2.2 in [BLP16] is accepted.

3.1.9 Constraint `sum`

This constraint is described in Section 4.1.4.1 in [BLP16]. Competition restrictions:

1. The condition is such that either the operator must be relational (i.e., must be in `{lt,le,gt,ge,eq,ne}`) and the (right) operand must be a value or a variable, or the operator must necessarily be `in` and the (right) operand must be an integer interval; See Section 1.5 in [BLP16].

3.1.10 Constraint `count`

This constraint is described in Section 4.1.4.2 in [BLP16]. Competition restrictions:

1. The element `<values>` can only contain (integer) values.
2. The condition is such that either the operator must be relational (i.e., must be in `{lt,le,gt,ge,eq,ne}`) and the (right) operand must be a value or a variable, or the operator must necessarily be `in` and the (right) operand must be an integer interval; See Section 1.5 in [BLP16].

3.1.11 Constraint nValues

This constraint is described in Section 4.1.4.3 in [BLP16]. Competition restrictions:

1. If present, the element <except> only contains one (integer) value.
2. The condition is such that the operator must necessarily be eq and the (right) operand must be a value or a variable.
3. Restricted forms (obtained by using the attribute restriction) are not accepted.

3.1.12 Constraint cardinality

This constraint is described in Section 4.1.4.4 in [BLP16]. Competition restrictions:

1. The element <values> can only contain (integer) values.
2. Restricted forms (obtained by using the attribute restriction) are not accepted.

3.1.13 Constraint maximum

This constraint is described in Section 4.1.5.1 in [BLP16]. Competition restrictions:

1. The condition is such that the operator must necessarily be eq and the (right) operand must be a value or a variable.
2. The element <index>, used for the variant <arg_max>, is not accepted.

3.1.14 Constraint minimum

This constraint is described in Section 4.1.5.2 in [BLP16]. Competition restrictions:

1. The condition is such that the operator must necessarily be eq and the (right) operand must be a value or a variable.
2. The element <index>, used for the variant <arg_min>, is not accepted.

3.1.15 Constraint element

This constraint is described in Section 4.1.5.3 in [BLP16]. Competition restrictions:

1. The optionnal attribute startIndex, if present, is necessarily equal to 0.
2. The attribute rank, whose default value is "any", cannot be present.

3.1.16 Constraint channel

This constraint is described in Section 4.1.5.4 in [BLP16]. Competition restrictions:

1. Restricted forms (obtained by using the attribute restriction) are not accepted.

3.1.17 Constraint `noOverlap`

This constraint is described in Section 4.1.6.2 in [BLP16]. Competition restrictions:

1. In case the element `<lengths>` contains (integer) values, whatever is the dimension, the value 0 is not accepted.

3.1.18 Constraint `cumulative`

This constraint is described in Section 4.1.6.3 in [BLP16]. Competition restrictions:

1. The element `<ends>` is not accepted.

2. The variant, using `<machines>`, is not accepted.

3.1.19 Constraint `instantiation`

This constraint is described in Section 4.1.8.2 in [BLP16]. There is no competition restriction for this constraint.

3.1.20 Meta-Constraint `slide`

This meta-constraint is described in Section 8.1 in [BLP16]. Competition restrictions:

1. Only one element `<list>` is accepted.

2. The constraint template must be of form `<intension>` or `<extension>`.

3.2 Constraints for Mini-solver Tracks

All general restrictions introduced for Standard tracks hold. Additionnally, the constraints that are accepted for Mini-solver tracks are restricted to five types, as decribed below.

3.2.1 Constraint `intension`

This constraint is described in Section 4.1.1.1 in [BLP16]. Competition restrictions for Mini-solver Tracks : only primitive constraints with one of the following form will be considered. In what follows, x, y and z denote integer variables, k denotes an integer value, $\odot$ denotes a relational operator in $\{<, \leq, \geq, >, =, \neq\}$ and $\oplus$ denotes a binary arithmetic operator in $\{+, -, *, /, \%, ||\}$, with $||$ being the distance.

- $x \odot k$ $\quad$ $k \odot x$
- $x \odot y$
- $(x \oplus k) \odot y$ $\quad$ $(k \oplus x) \odot y$ $\quad$ $x \odot (y \oplus k)$ $\quad$ $x \odot (k \oplus y)$
- $(x \oplus y) \odot y$ $\quad$ $x \odot (y \oplus z)$

3.2.2 Constraint `extension`

This constraint is described in Section 4.1.1.2 in [BLP16]. Competition restrictions for Mini-solver Tracks (the same as for Standard Tracks):

1. Short tables (i.e., tables with tuples containing '*') and compressed tables (i.e., tables with compressed tuples) are not accepted.

2. Empty Tables (i.e., tables with with 0 support or 0 conflict) are not accepted.

Note that unary, binary and n-ary extensional constraints are accepted.

3.2.3 Constraint `allDifferent`

This constraint is described in Section 4.1.3.1 in [BLP16]. Competition restrictions for Mini-solver Tracks:

1. The element `<except>` cannot be present.

2. Restricted forms (obtained by using the attribute `restriction`) are not accepted.

No advanced form of `allDifferent`, as e.g., `allDifferent-matrix`, is accepted.

3.2.4 Constraint `sum`

This constraint is described in Section 4.1.4.1 in [BLP16]. Competition restrictions for Mini-solver Tracks:

1. The condition is such that the operator must be relational (i.e., in `{lt,le,gt,ge,eq,ne}`) and the (right) operand must be a value or a variable; see Section 1.5 in [BLP16].

3.2.5 Constraint `element`

This constraint is described in Section 4.1.5.3 in [BLP16]. Competition restrictions for Mini-solver Tracks:

1. The optionnal attribute `startIndex`, if present, is necessarily equal to 0.

2. The element `<index>` is necessarily present but the attribute `rank`, whose default value is `"any"`, cannot be present.

4 Resources: Benchmarks and Tools

Many benchmarks can be found at:

www.xcsp.org/series

The organizers invite *anybody* to submit new benchmarks. The organizers are particularly interested in new problem instances originating from real-world applications. For generating new XCSP3 instances, one can use the Java-based modeling API, called MCSP3. See its description in this document; once a model is developed, it is easy to generate XCSP3 instances by compiling it while furnishing data.

Some tools are also provided. They can be found at:

www.xcsp.org/tools

Currently, you can find:

- a C++ parser
- a Java parser
- a tool for checking solutions and costs
- a Java-based modeling API

5 Execution Environment

Solvers will run on a cluster of computers using the Linux operating system. They will run under the control of another program (called runsolver) that will enforce some limits on both used memory and total CPU time. Solvers will be run inside a sandbox that will prevent unauthorized use of the system (network connections, file creation outside the allowed directory, among others).

Solvers can be run as either 32 bits or 64 bits applications. If you submit an executable, you are required to provide us with an ELF executable (preferably statically linked). Authors submitting solvers in source form will have to specify if it should be compiled in 32 bits or 64 bits mode.

Two executions of a solver with the same parameters and system resources are expected to output the same result in approximately the same time (so that the experiments can be repeated).

5.1 Command Line

During the submission process, you will be asked to provide the organizers with a suggested command line that should be used to run your solver. In this command line, you will be asked to use the following placeholders, which will be replaced by the actual information by the evaluation environment.

- BENCHNAME will be replaced by the name of the file containing the XCSP3 instance to solve. Obviously, the solver must use this parameter or one of the following variants: BENCHNAMENOEXT (name of the file with path but without extension), BENCHNAMENOPATH (name of the file without path but with extension), BENCHNAMENOPATHNOEXT (name of the file without path nor extension).
- RANDOMSEED will be replaced by a random seed which is a number between 0 and 4294967295. This parameter MUST be used to initialize the random number generator when the solver uses random numbers. It is recorded by the evaluation environment and will allow to run the program on a given instance under the same conditions if necessary.
- TIMELIMIT (or TIMEOUT) represents the total CPU time (in seconds) that the solver may use before being killed. May be used to adapt the solver strategy.

- MEMLIMIT represents the total amount of memory (in MiB) that the solver may use before being killed. May be used to adapt the solver strategy.
- NBCORE will be replaced by the number of processing units that have been allocated to the solver. Note that, depending on the available hardware, a processing unit may be either a processor, a core of a processor or a "logical processor" (in hyper-threading).
- TMPDIR is the name of the only directory where the solver is allowed to read/write temporary files
- DIR is the name of the directory where the solver files will be stored

Examples of command lines:

```
DIR/mysolver BENCHNAME RANDOMSEED
DIR/mysolver --mem-limit=MEMLIMIT --time-limit=TIMELIMIT --tmpdir=TMPDIR BENCHNAME
java -jar DIR/mysolver.jar -c DIR/mysolver.conf BENCHNAME
```

As an example, these command lines could be expanded by the evaluation environment as:

```
/solver10/mysolver /tmp/zebra.xml 1720968
/solver10/mysolver --mem-limit=900 --time-limit=1200 --tmpdir=/tmp/job12345 /tmp/zebra.xml
java -jar /solver10/mysolver.jar -c /solver10/mysolver.conf /tmp/zebra.xml
```

The command line provided by the submitter is only a suggested command line. Organizers may have to modify this command line (e.g., memory limits of the Java Virtual Machine (JVM) may have to be modified to cope with the actual memory limits).

The solver may also (optionally) use the values of the following environment variables:

- TIMELIMIT (or TIMEOUT) (the number of seconds it will be allowed to run)
- MEMLIMIT (the amount of RAM in MiB available to the solver)
- TMPDIR (the absolute pathname of the only directory where the solver is allowed to create temporary files)

After TIMEOUT seconds have elapsed, the solver will first receive a SIGTERM to give it a chance to output the best solution it found so far (in the case of an optimization problem). One second later, the program will receive a SIGKILL signal from the controlling program to terminate the solver.

The solver cannot write to any file except standard output, standard error and files in the TMPDIR directory. A solver is not allowed to open any network connection or launch unexpected external commands. Solvers may use several processes or threads. Children of a solver process are allowed to communicate through any convenient means (Pipes, Unix or Internet sockets, IPC, ...). Any other communication is strictly forbidden. Solvers are not allowed to perform actions that are not directly related to the resolution of the problem.

5.2 Output Format

To communicate their answers, solvers must print messages to the standard output and those messages will be used to check the results. The first two characters of a line allow us to classify it into different categories, which indicate the meaning of the line. With the exception of "o " lines, there is no specific order imposed on the lines output by solvers.

- **status line**
  This line starts by the two characters: lower case s followed by a space (ASCII code 32). Only one such line is allowed, and it is mandatory. This line gives the answer of the solver. It must be one of the following answers:

  - `s UNSUPPORTED`
    This line should be printed by the solver when it discovers that the XCSP3 instance contains a non-supported feature. As an example, a solver that cannot deal with a global constraint should print this line when such a constraint is present.
  - `s SATISFIABLE`
    This line indicates that the solver has found a solution, and in such a case, a "v " line (see below) is mandatory. For CSP, this line must be printed when the solver has found a solution. For COP, this line must be in the output when the solver has found a solution that it couldn't prove to be optimal.
  - `s OPTIMUM FOUND`
    This line must be printed when the solver has found an optimal solution for a COP instance, and in such a case, a "v " line (see below) is mandatory. This answer implies that the solver has proved that no other solution can give a better value of the objective function. This answer must not be used for CSP instances.
  - `s UNSATISFIABLE`
    This line must be output when the solver can prove that the instance has no solution.
  - `s UNKNOWN`
    This line must be output in any other case, i.e. when the solver is not able to tell anything about the instance.

  It is of uttermost importance to respect the exact spelling of these answers. Any mistake in the writing of these lines will cause the answer to be disregarded.

  Solvers are not required to provide any specific exit code corresponding to their answer.

  If the solver does not output a status line, or if the status line is misspelled, then UNKNOWN will be assumed.

- **values line**
  This line starts by the two characters: lower case v followed by a space (ASCII code 32). It is mandatory when the instance is satisfiable. More than one "v " line is allowed but the evaluation environment will act as if their content was merged.

  If the solver finds a solution (i.e., if the solver outputs "s SATISFIABLE" or "s OPTIMUM FOUND"), it must provide a solution. For CSP or COP, this solution is an instantiation that satisfies every constraint. For COP, this instantiation must be such that the value of the objective function corresponds to the best one that the solver was able to find.

Solutions must respect the format described in Section 2.11 of [BLP16]. However, it is important to note that the attributes `type` and `cost` that can be associated with the element `<instantiation>` are **not required** in the context of the competition. These attributes, if present, will simply be ignored.

Importantly, the solution can be output on several successive "v " lines, provided that each "v " line must be terminated by a Line Feed character (the usual Unix line terminator '\n'). A "v " line that does not end with that terminator will be ignored because it will be considered that the solver was interrupted before it could print a complete solution.

As an illustration, the following output is valid for the COP instance (Example 4) given in Chapter 1 of [BLP16]:

```
v <instantiation type="optimum" cost="1700">
v   <list> b c </list>
v   <values> 2 2 </values>
v </instantiation>
```

and the following output is valid for the CSP instance (Example 25) given in Section 2.11 of [BLP16]:

```
v <instantiation type="solution">
v   <list> x[] </list>
v   <values> 1 1 2 * </values>
v </instantiation>
```

As the attributes `type` and `cost` are not required (and simply ignored by our environment), we could have written:

```
v <instantiation>
v   <list> b c </list>
v   <values> 2 2 </values>
v </instantiation>
```

and

```
v <instantiation>
v   <list> x[] </list>
v   <values> 1 1 2 * </values>
v </instantiation>
```

- **objective line**
  These lines start by the two characters: lower case o followed by a space (ASCII code 32). **These lines are mandatory for incomplete solvers.** As far as complete solvers are concerned, they are not strictly mandatory but solvers are strongly invited to print them. These lines are only relevant for COP instances.

  Whenever the solver finds a solution with a better value of the objective function, it is asked to print an "o " line with the current value of the objective function. Therefore, an "o " line must contain the lower case o followed by a space and then by an integer that represents the better value of the objective function. "o " lines should be output as soon as the solver finds a better solution and be ended by a standard Unix end of line character ('\n'). Programmers are advised to flush immediately the output stream. As an example; let us consider Example 2 in Chapter 1 of [BLP16]. Let us assume that the solver finds first this solution:

```
<instantiation id='sol1' type='solution' cost='450'>
  <list> b c </list>
  <values> 0 1 </values>
</instantiation>
```

and later:

```
<instantiation id='sol2' type='solution' cost='1700'>
  <list> b c </list>
  <values> 2 2 </values>
</instantiation>
```

which is finally proved to be optimal by the solver. The output by the solver can be (using this time only one "v " line):

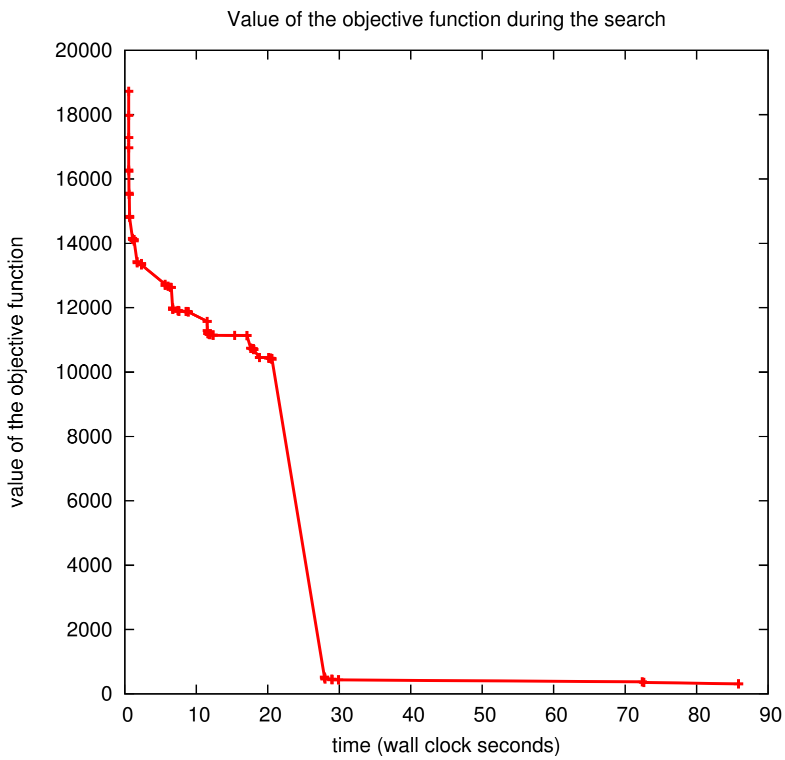
```
o 450
o 1700
s OPTIMUM FOUND
v <instantiation> <list> b c </list> <values> 2 2 </values> </instantiation>
```

The evaluation environment will automatically timestamp each of these lines so that it is possible to know when the solver has found a better solution and how good the solution was. The goal is to be able to analyze the way solvers progress toward the best solution. As an illustration, here is a sample of the output of a solver, with each line timestamped (first column, expressed in seconds of wall clock time since the start of the program).

```
0.00     c Time Limit set via TIMEOUT to 1800
0.51     c Initial problem consists of 6774 variables and 100 constraints.
0.55     c    preprocess terminated. Elapsed time: 0.45
0.55     c Initial Lower Bound: 0
0.63     o 235947
0.63     o 226466
0.63     o 217758
0.75     o 186498
1.16     o 178319
2.42     o 168389
3.13     c Restart #1 #Var: 6774 LB: 0 @ 3.03
4.89     c Restart #2 #Var: 6774 LB: 0 @ 4.79
5.73     o 160358
6.44     o 159206
7.52     o 150077
9.09     o 149533
12.14    o 140853
17.74    o 140264
19.61    o 131636
29.81    o 15450
34.00    o 7066
41.66    o 5000
84.01    o 3905
```

```
84.01     c NEW SOLUTION FOUND: 3905 @ 83.873
84.61     s OPTIMUM FOUND
84.61     v ... // solution not shown here for space reasons
84.61     c Total time: 84.478 s
```

and here is an example of graph which can be generated from such 'o ' lines:



- **diagnostic line**
  These lines are optional and start with the two following characters: lower case `d` followed by a space (ASCII code 32). Then, a keyword followed by a value must be given on this line.

  More precisely, a *diagnostic* is a (name,value) pair that gives an information about the work carried out by the solver. As indicated above, each diagnostic is a line of the form '`d NAME value`', where `NAME` is a sequence of letters describing the diagnostic, and `value` is a sequence of characters defining its value. The following diagnostic is predefined:

  `WRONG_DECISIONS`: The total number of wrong decisions which have been carried out (as defined in [**?**]).

  Contestants wishing to record other diagnostics than the one listed before above should inform the organizers.

- **comments line**
  A line which is not one the special lines defined above, or which explicitly starts with the two characters: lower case c followed by a space (ASCII code 32) is a comment line,

and is ignored. These lines are thus optional and may appear anywhere in the solver output.

They contain any information that authors want to emphasize, such as #backtracks, #flips,... or internal CPU time. They are recorded by the evaluation environment for later viewing but are otherwise ignored. At most one megabyte of solver output will be recorded. So, if a solver is very verbose, some comments may be lost.

Submitters are advised to avoid printing comment lines which may be useful in an interactive environment but otherwise useless in a batch environment. For example, printing comment lines with the number of constraints read so far only increases the size of the logs with no benefit.

If a solver is really too verbose, the organizers will ask the submitter to remove some comment lines.

5.3 Special Considerations for Incomplete Solvers

Complete solvers are solvers which can always decide the satisfiability of a CSP instance and the optimality of a COP instance, provided that enough time and memory are given. Incomplete solvers may loop endlessly in a number of cases; local search algorithms are examples of incomplete solvers. Both kinds of solvers are welcome in this competition. Submitters will have to indicate if their solver is complete or incomplete on the submission form.

5.3.1 Complete solvers

There is no special requirement about complete solvers. See the input and output format that all solvers must respect for details.

5.3.2 Incomplete solvers

Incomplete solvers are definitely welcome in the competition.

For CSP, an incomplete solver will stop as soon as it finds a solution and will time out if it can't find one. The only difference with a complete solver is that it will time out systematically on unsatisfiable instances.

For COP, an incomplete solver will systematically time out because it will be unable to prove that it has found the optimum solution. Yet, it may have found the optimum value well before the time out. In order to get relevant information in these categories, an incomplete solver must fulfill two requirements:

1. it must intercept the SIGTERM signal sent to the solver on timeout and output either "s UNKNOWN" or "s SATISFIABLE" with the "v " line(s) corresponding to the best solution it has found

2. it MUST output an "o " line whenever it finds a better solution so that, even if the solver always timeouts, the timestamp of the last "o " line indicates when the best solution was found. Keep in mind that it is the evaluation environment which is in charge of timestamping "o " lines.

5.4 Special Considerations for Parallel Solvers

The execution environment will bind the solvers to a subset of all available processing units. The environment variable NBCORE will indicate how many processing units have been granted to the solver. The solver will not have access to more processing units than NBCORE. This implies that if the solver uses x threads or processes (with $x >$NBCORE), $x-$NBCORE threads or processes will necessarily sleep at one time.

As an example, if the competition is run on hosts with 2 quad-core processors (8 cores in total), several scenarios are possible:

- one single solver is run on the host, it is allowed to use all 8 cores (NBCORE=8).
- two solvers are run simultaneously, each one being assigned to a given processor (which means that a solver is assigned 4 cores, hence NBCORE=4).
- 4 solvers are run simultaneously, each one being assigned to a fixed set of 2 cores (belonging to the same CPU), hence NBCORE=2.
- more generally, a single solver may be assigned any number x of cores (from 1 to 8 in this example) to simulate the availability of x processing units.

The solver might use the NBCORE environment variable to adapt itself to the number of available processing units.

A solver must not modify its processor affinity (calls to `sched_setaffinity(2)` or `taskset(1)`) to get access to a processing unit that was not initially allocated to the solver. It may however modify its processor affinity to use a subset of the initially allocated processing units.

6 Entering the Competition

Contestants can enter the competition with one or two solvers per track. Contestants are expected to submit their solver(s) and contribute some instances (as many instances as wished). Submitted instances will be made available on the evaluation web site shortly after the actual beginning of the competition. We cannot accept benchmarks which cannot (for various reasons) be publicly available (because anyone must be able to reproduce the experiments of the competition). Each contestant will have the possibility to select 5 instances (that can be kept hidden) with the guarantee that they will be used for the competition. They will also have to submit a position paper (at least 2 pages) in a second stage.

Of course, we expect that contestants propose solvers that recognize XCSP3 (either natively or by embedding a conversion procedure).

The deadline for submitting both benchmarks and solvers is June 10, 2017. Submission of solvers and benchmarks will be possible online in May 2017 at http://www.cril.fr/CompetitionXCSP17/.

7 Ranking

Basically, solvers will be ranked on the number of times a solver is able to give the best answer obtained during the competition. Ties will be broken on the cumulated CPU/wall-clock time to give these answers. Other ranking schemes may be introduced to help identify remarkable features.

Special Style Prize for Mini-solvers. In addition to classical ranking (i.e., ranking of solvers based on relative efficiency), a special prize will be awarded to the Mini-solver(s) with the most stylish code. Stylish must be understood here as "highly readable, well-structured, well-documented, and easily extendable". The jury will select the Mini-solver(s) for this prize, while paying attention that the Mini-solver is reasonably efficient.

Wrong Answers. Note that a solver is declared to give a wrong answer in the following cases:

- It outputs UNSATISFIABLE for an instance which can be proved to be satisfiable.
- For CSP and COP, it outputs SATISFIABLE or OPTIMUM FOUND, but provides an instantiation that does not satisfy every constraint. The only exception is when the solver outputs an incomplete "v " line (which does not end by '\n') in which case it is assumed that the solver was interrupted before it could output the complete model and the answer will be considered as UNKNOWN.
- It outputs OPTIMUM FOUND but there exists an instantiation with a better value of the objective function/cost that the one corresponding to the printed solution.

When a solver provides even one single wrong answer in a given track, the solver's results in that track will be excluded from the final evaluation results because they cannot be trusted. Exceptionally, the organizers may decide to present separately the results of such a solver but only if it obtained particularly good results and if a detailed explanation of the problem as well as a correction is provided by the submitters.

A solver that ends without giving any solution, or just crashes for some reason (internal bugs...), is simply considered as giving an UNKNOWN result.

8 Committees

8.1 Organization

People that are in charge of running the competition come from several institutions:

CRIL	Christophe Lecoutre, Cédric Piette and Olivier Roussel
ICTEAM	Pierre Schauss
I3S	Arnaud Malapert
LS2N	Charles Prudhomme

They can be reached at `compet <at> xcsp.org`.

8.2 Judges

The competition jury is composed of three judges who are in charge of taking decisions when rules are unclear and of validating the results of the competition. The selection of instances used in the competition is also managed by the judges (who can possibly assign an independent committee for this task, if wished). The three judges are:

- Claude-Guy Quimper from Université Laval, Québec, Canada
- Helmut Simonis from Insight Centre for Data Analytics, Cork, Ireland

- Christine Solnon from INSA, Lyon, France

For the 2017 XCSP3 competition, the jury has the responsibility of selecting:

- for the standard tracks
  - a set of 500 CSP instances, and
  - a set of 500 COP instances
- for the mini-solver tracks
  - a set of 100 CSP instances, and
  - a set of 100 COP instances

All the selected instances will be classified in series, the size of which will typically range from 5 to 20. It is expected to have easy, medium and hard instances within each series, in order to be able to observe the scaling behavior of solvers.

Concerning the selection, the jury can proceed independently as follows for each series:

- select precisely a subset of instances either from a series that is available at `www.xcsp.org` (for example, a precise list of 10 filenames for problem `Langford` in series `m1/k2`) or from a series proposed by a participant.
- give the number k of instances to be selected either from a series that is available at `www.xcsp.org` or from a series proposed by a participant. The k instances will be randomly selected by using an auxiliary procedure (which just requires as input a seed chosen by the judges).
- possibly develop new original series of instances (for example, by using the Java-based modeling API MCSP3).

We recall that anybody is welcome to submit new benchmarks (to the judges), and that each participant has the possibility of selecting exactly 5 instances (while proposing many more instances that may finally be selected or not), and must inform the jury of his/her selection.

References

[BLP16] F. Boussemart, C. Lecoutre, and C. Piette, *XCSP3: An integrated format for benchmarking combinatorial constrained problems*, Tech. Report arXiv:1611.03398, CoRR, 2016, Available from https://arxiv.org/abs/1611.03398 and http://www.xcsp.org/format3.pdf.