Call for Solvers and Benchmarks
Fourth International Constraint Solver Competition
(CSP, Max-CSP and Weighted-CSP competition)
`http://cpai.ucc.ie/09/`

The Fourth International Constraint Solver Competition (`CSC'2009`) is organised to improve our knowledge of what is behind the efficiency of constraint satisfaction algorithms, heuristics, solving strategies, and constraint systems. `CSC'2008`, the third edition of the competition, considered the **CSP** and the **Max-CSP** and problem instances consisting of binary and non-binary, extensional and intensional constraints, as well as a few global constraints. `CSC'2009` further extends the scope by introducing the Weighted-CSP (**WCSP**) problem and by allowing *any* global constraint defined on integer variables, provided the constraint has been requested by some contestant. To propose a direction for contestants, the competition organizers have selected 10 central global constraints (eight of them identified by Nicolas Beldiceanu). A list of these constraints is provided on Page 6. However, solvers will also be evaluated for other kinds of contestant-proposed global constraints that they support. As already indicated, constraints are only used if they are supported by some contestant's solver. To indicate that their solver supports a given global constraint contestants should formally *request* the global constraint and submit at least 20 problem instances with this global constraint.

The changes with respect to `CSC'2008` have some implication on the ranking of the solvers. This is explained in Section 9.

As a summary, `CSC'2009` will consider **CSP**, **Max-CSP** and **WCSP** problems, constraints defined in extension, constraints in intension, and *any* kind of global constraints defined on integer variables.

*To participate to the competition, it is not necessary to submit a solver which can deal with all kinds of constraints and all problems. Submitting a solver which is capable of dealing with only one kind of constraint (e.g. binary, extensional constraints) and one kind of **CSP**(e.g. ordinary **CSP**) is also allowed. The only requirement is that the solver must indicate when it has no support for a given kind of constraint.*

The remainder of this call for solvers and benchmarks presents (1) the problems and categories that will be considered during the competition, (2) details about the representation of **CSP** instances, (3) the execution environment, and (4) the rules.

A printable version of this call may be found at `http://cpai.ucc.ie/09/call2009.pdf`. A short version may be found at `http://cpai.ucc.ie/09/call2009`.

Contents

1 Timetable

The deadlines of the competition are as follows:

Opening of the registration site at http://www.cril.univ-artois.fr/CPAI09	May 2009
Constraint request and submission deadline	April 10, 2009
Pre-registration of contestants	May 30, 2009
Final registration (submission of solvers and benchmarks)	June 10, 2009
Test of solvers conformance	June 2009
Competition running	Summer 2009
Final results available	during CP 2009

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 in June 2009. Bug fixes will be due one week after the bug has been reported to the authors.

2 Requesting and Submitting Global Constraints

It is recalled from the introduction that this year's competition allows *any* global constraint defined on integer variables, provided the constraint has been formally requested by some contestant team. Any contestant team requesting a global constraint is expected to *submit* at least 20 instances that involve the global constraint. Binary, non-binary, extensional and intensional constraints should not be requested.

To request their global constraints, contestant teams are invited to send an email to cspcomp⟨at⟩cril.univ-artois.fr. Constraints may be requested until April 10, 2009. A separate email should be sent for each global constraint. The subject of the email should read "GLOBAL CONSTRAINT REGISTRATION: ⟨name of the global constraint⟩" and the body of the email should describe the name of the requesting contestant team and a URL where to find the team's submitted problem instances with the global constraint. It is recalled that there should be at least 20 such instances. The name and semantics of the global constraint must correspond to the definition of the global constraint catalog (http://www.emn.fr/x-info/sdemasse/gccat/).

Details of the specification format for global constraints may be found in the document entitled *XML Representation of Constraint Networks* which may be downloaded from http://www.cril.fr/~lecoutre/research/benchmarks/. This document also presents examples.

3 Changes Compared to Previous Editions

This section describes the differences between the Fourth International Constraint Solver Competition (`CSC'2009`) and the Third International Constraint Solver Competition, which was organised in 2008.

`CSC'2009` allows *any* kind of global constraints defined on integer variables. Since one of the goal of the competition is to accept any kind of solver, we don't expect each submitted solver to support all global constraints, or indeed all constraints.

Having solvers with different constraint handling capabilities, we need a mechanism to determine the constraints which the solvers can handle. To do this we introduce the notion of a solver's *Boolean capability vector*, capability vector for short, which consists of a sequence of ones and zeros acting as indicators of the constraints which the solver can handle. Solvers which have the same capability can be naturally compared. Solvers with different capabilities can be compared on instances which belong to the intersection of their capabilities, provided it is nonempty.

The first step in the competition is to collect the capabilities of each solver. To determine the solver's capability vector, we require that solver is capable of outputting a special answer '`s UNSUPPORTED`' whenever it reads a constraints which it cannot handle. This should be quite easy to implement in each solver. For example, it may be generated by a driver script which generates this output if the real solver fails to parse an instance.

Two solvers which have a nonempty capability vector intersection can be ranked on the basis of their results for instances belonging to the intersection of their capabilities. Solvers will be clustered according to their capabilities and a ranking will be established between solvers which support the same kinds of constraints. The goal of the competition is to produce all relevant ranking. However, there is potentially an exponential number of different capabilities and therefore an exponential number of rankings. To restrict this number, rankings will not be established when a cluster of solvers contains fewer than 3 solvers (with different authors) or when there are fewer than 20 instances specific to the cluster. Should the number of rankings still be too large, the organisers retain the right to ask the independent judges to select a limited number of relevant rankings to publish.

For example, let's assume that solvers BE_i have support for only binary constraints in extension, solvers NG_i have support for all constraints but global ones and solvers AD_i have support for *allDifferent* as well as any other kind of non global constraint. Let's further assume that a few other solvers have the same capabilities. The competition will produce a ranking for

- the cluster of solvers with support for *allDifferent* as well as any other kind of non global constraint. This will gather solvers AD_i.

- the cluster of solvers with support for all constraints but global ones. It will compare solvers AD_i and NG_i.

- the cluster of solvers with support for only binary constraints in extension. It will gather solvers AD_i, NG_i and BE_i

In a cluster of solvers, a solver which would answer 's UNSUPPORTED' on even a single instance excludes itself from the ranking because it indicates that it doesn't have support for all kinds of constraints of the cluster. This solver will still appear in the sub clusters where it never answers 's UNSUPPORTED'.

If in a cluster of solvers, a solver gives an incorrect answer (UNSATISFIABLE on a satisfiable instance for example), it is reported as incorrect in this cluster and excluded from the ranking. This solver will still appear in the ranking of the sub clusters as long as it doesn't produce an incorrect answer.

All global constraints are allowed in the instances. However, if no instance with a given global constraint is submitted to the competition, this global constraint will be ignored. In other words, organisers will not ensure that any possible global constraint is used in the competition.

The following is the list of the 10 global constraints selected to give a direction to the contestants.

- *allDifferent*;

- *cumulative*;

- *weightedSum*;

- *element*;

- *disjunctive*;

- *lexLess*;

- *lexLessEq*;

- *nValue*;

- *globalCardinalityWithCosts*; and

- *globalCardinality*.

4   Problems and Categories

Anyone can submit a solver to any particular solver category with respect to a general problem (**CSP**, **Max-CSP** or **WCSP**). For example, it is possible to just register a complete solver for **CSP**. It is no more necessary to register to specific instance categories (e.g. the category of instances involving binary extensional constraints) because one new requirement about solvers is to output 's UNSUPPORTED' when a constraint not supported by the solver is encountered at loading.

4.1 Problems

There are three problems:

- **CSP**;

- **Max-CSP**; and

- **WCSP**.

The objective for **CSP** is finding a solution or proving that no solution exists. The objective for **Max-CSP** is finding an instantiation of variables that violates as few constraints as possible (or, equivalently, that satisfies as many constraints as possible). The objective for **WCSP** is finding an instantiation of variables that corresponds to the minimal violation cost (a cost or weight is associated with each tuple of each constraint). **CSP** is a decision problem whereas **Max-CSP** and **WCSP** are optimisation ones.

4.2 Solver Categories

The solver categories are:

- complete

- incomplete

Complete solvers can determine if an instance is satisfiable or not (or find and prove the optimum for **Max-CSP** and **WCSP**) whereas incomplete solvers cannot prove the unsatisfiability or the optimum.

5 Resources

5.1 Format

The description of the XCSP 2.1 format used to represent **CSP** instances can be found at http://cpai.ucc.ie/09/ or http://www.cril.univ-artois.fr/CPAI09/.

5.2 Benchmarks

Many benchmarks (including those used for the 2006 and 2008 **CSP** Solver Competition) may be found at:

> http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/

The organisers invite *anybody* to submit new benchmarks. The organisers are particularly interested in new problem instances originating from real-world applications.

5.3   Tools

Some tools are also provided. They can be found from `http://cpai.ucc.ie/09/`.
`C++` and `C` parsers for the `XML` format can be found at:

> `http://www.cril.univ-artois.fr/~roussel/CSP-XML-parser`

A Java parser for the `XML` format can be found at:

> `http://www.cril.univ-artois.fr/~lecoutre/research/tools`

Other tools that are currently available from the last URL given above are:

- *SolutionChecker*: a tool that allows to compute the number of constraints violated by a full instantiation of the variables of a **CSP** instance.

- *InstanceChecker*: a tool that allows to check the validity of **CSP** instances (in format `XCSP 2.1`) and to convert in extension constraints defined in intension.

- *InstanceShuffler*: a tool that allows to shuffle variables and constraints of **CSP** instances.

New tools that have been developed (by Andrea Rendl) are:

- *xcsp2ep*: a converter from `XCSP` to `Essence'`.

- *tailor*: a converter from `XCSP`/`Essence'` to `Minion`.

- *ep2xcsp*: a converter from `Essence'` to `XCSP`.

6   Execution Environment

Solvers will run on a cluster of computers using the Linux operating system. They will run under the control of another program (runsolver) which will enforce some limits on the memory and the total CPU time used by the program. Solvers will be run inside a sandbox that will prevent unauthorised use of the system (network connections, file creation outside the allowed directory, among others). Constestants can choose to submit a 32 or 64 bits application. Executables should be submitted as an `ELF` executable (preferably statically linked). Submission of source files must indicate how to compile the solver. Two executions of a solver with the same parameters and system resources must output the same result in approximately the same time (so that the experiments can be repeated).

During the submission process, you will be asked to provide the organisers 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 given by the evaluation environment.

BENCHNAME: will be replaced by the name of the file containing the instance to solve (including the path to the file). Obviously, the solver must use this parameter (or alternatively `BENCHNAMENOEXT`).

BENCHNAMENOEXT: will be replaced by the base name of the file containing the instance to solve (i.e. without the filename extension, but with the complete path to the file).

RANDOMSEED: will be replaced by a random seed which is a number between 0 and 4,294,967,295. This parameter MUST be used to initialise 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.

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.

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

The command line provided by the submitter is only a suggested command line. Organisers 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:

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` signal to give it a chance to output the best solution found so far. (This is useful for optimisation problems.) One second later, the program will receive a `SIGKILL` signal from the controlling program to terminate the solver.

Similarly, a solver that uses more memory than the limit defined by `MEMLIMIT` will be sent a `SIGTERM` followed one second later by a `SIGKILL`.

The time and memory limits will be defined in accordance with the number of solvers and benchmarks that will enter the competition. One should expect a time limit of at least 20 minutes and a memory limit of at least 900 MiB.

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 external commands which are not related to the solving process. Solvers can use several processes or threads.

7 Output Rules

The evaluation environment records everything that is output by your solver on stdout/stderr (up to a limit of 1MiB) and is able to timestamp each line. This can be very informative to check how your solvers behaved on some instances.

Therefore solvers must output messages to the standard output and those messages will be used to check the results. The output format is inspired by the `DIMACS` output specification of the `SAT` competition and can be used to manually check some results. Lines output by the solver should be prefixed by 'c ', 's ', 'v ', 'd ' or 'o '. Lines which do not start with one of these prefixes are considered as comment lines and are ignored. The meaning of these prefixes is detailed below.

7.1 Lines

There exist 5 different types of lines. They are defined as follows:

- solution ('s ' line)

  These lines are mandatory and start with the two following characters: lower case `s` followed by a space (ASCII code 32). These two characters are followed by one of the following answers:

  UNSUPPORTED: for **CSP**, **Max-CSP** and **WCSP**.

  SATISFIABLE: for **CSP**, **Max-CSP** and **WCSP**.

  UNSATISFIABLE: for **CSP**.

  UNKNOWN: for **CSP**, **Max-CSP** and **WCSP**.

  OPTIMUM FOUND: for **Max-CSP** and **WCSP**.

It is of utmost 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.

It is important to note that UNSUPPORTED is used when the solver recognises a constraint that is not implemented.

See Subsections 7.2 and 7.3 about their use.

- values ('v ' line)

  These lines are mandatory and start with the two following characters: lower case v followed by a space (ASCII code 32) and followed by a solution of the problem.

  Here, a solution is a sequence of numbers (only), which are separated by one or more space symbols. The ith number in the solution corresponds to the "assignment" to the ith variable of the problem in the *original* problem specification in standard input format. (Note that this may have consequences for solvers that introduce auxiliary variables, rename variables, or change their order.)

- diagnostic ('d ' 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. See section 7.4 for details about the diagnostics.

- comment ('c ' line)

  Such lines are optional and start with the two following characters: lower case c followed by a space (ASCII code 32). These lines are optional and may appear anywhere in the solver output. They contain any information that authors want to output. 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 outputting comment lines which may be useful in an interactive environment but otherwise useless in a batch environment. For example, outputting 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 organisers will ask the submitter to remove some comment lines.

- objective cost ('o ' line) (for **Max-CSP** and **WCSP**, only)

  These lines start with the two following characters: lower case o followed by a space (ASCII code 32). These two characters are followed by one integer.

  See Subsection 7.3 about its use.

Important: Any line must be terminated by a Line Feed character: the usual Unix line terminator '\n' (ASCII code 10). A 'v ' line which does not end with that terminator will be ignored because it will be considered that the solver was interrupted before it could output a complete solution. Also, it is important that you don't forget to flush the output as soon as you have printed a 's ' line or a 'v ' line.

7.2 Specific rules for CSP Solvers

A **CSP** solver must output exactly one 's ' line (it is mandatory) and in addition, when the instance is found SATISFIABLE, exactly one 'v ' line. These lines are not necessarily the first ones in the output since the **CSP** solver can output some 'c ' and 'd ' lines in any order. For a **CSP** solver, the 's ' line must correspond to one of the following answers:

s UNSUPPORTED: when the solver recognises an instance with a constraint not implemented;

s SATISFIABLE: when the solver has found a solution;

s UNSATISFIABLE: when the solver can prove that the instance has no solution;

s UNKNOWN: when the solver is not able to tell anything about the instance.

If the solver does not output a 's ' line, or if the 's ' line is misspelled, then UNKNOWN will be assumed. For a **CSP** solver, the 'v ' line provides a valuation of each variable that satisfies all constraints. This will be used to check the correctness of the answer.

7.3 Specific rules for Max-CSP and WCSP solvers

Since a **Max-CSP** or **WCSP** solver will not stop as soon as it finds a solution but instead will try to find a better solution, it must be given a way to output the best solution it found even when it reaches the time limit. There are two options depending on your ability to intercept signals. The first option may be used if your solver is able to catch the signal SIGTERM which will be sent by the evaluation environment at the time out and just one second before the solver is actually killed by a SIGKILL. This is the preferred option. The second option should be used when the first option is inapplicable.

1. You can intercept signals:

   Whenever your **Max-CSP** or **WCSP** solver finds a better solution (or simply the first one) during search, it can output an 'o ' line with the current cost (number of unsatisfied constraints for **Max-CSP**) (these lines are not mandatory, but it is strongly advised to output them). **Important:** For **Max-CSP**, this is the number of unsatisfied constraints that must be output and not the number of satisfied constraints (as in 2006). Then, if your solver finds an optimal solution and proves its optimality (i.e., it can prove that no other assignment of the variables will provide a lower cost than this one) then it must output a 's ' line with OPTIMUM FOUND,

followed by a 'v ' line containing the optimal solution. If your solver is interrupted, then it must output a 's ' line with SATISFIABLE, followed by a 'v ' line containing the best found solution (keep in mind that you have only one second to do this).

This option saves some time as the solver avoids to output a certificate for each solution it found. It only outputs a certificate for the best solution which it was able to find.

Examples of C/C++ code to intercept the SIGTERM signal are available on the PB06 web site: http://www.cril.univ-artois.fr/PB06/coding.html. Although this page gives code which is not directly suitable for the **CSP** competition, it should be straightforward to adapt it.

2. If you can't (or don't want to) catch the SIGTERM signal:

   Then all you have to do is to output a 's ' line with SATISFIABLE when the first solution is found, and a certificate 'v ' line each time you find a solution which is better than the previous ones accompanied (this is mandatory) with an 'o ' line. Only the last complete certificate will be taken into account. If eventually, your solver proves that the last solution that was output is optimal, then it must output 's OPTIMUM FOUND'.

Example: a **Max-CSP** solver (that can intercept signal) first finds a solution with 19 unsatisfied constraints, then finds another solution with 16 unsatisfied constraints, and finally, a solution with 1 unsatisfied constraint. Some time later, it proves that no better solution exists and it outputs 's OPTIMUM FOUND' followed by the solution which only violates 1 constraint. The output of this solver will be:

```
o 19
o 16
o 1
s OPTIMUM FOUND
v 1 4 7 8 3 4
```

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 the cost of the solution. The goal is to analyse how solvers progress toward the best solution. The timestamped output will be for example:

```
0.57    o 19
1.23    o 16
2.7     o 1
10.5    s OPTIMUM FOUND
10.51   v 1 4 7 8 3 4
```

13

The first column in this example is the time at which the line was output by the solver (expressed in seconds of wall clock time since the beginning of the program).

A solver which doesn't intercept the SIGTERM signal may output for the same problem

```
c Got a first solution !
s SATISFIABLE
o 19
v 1 1 1 1 1 1
c Found a better solution
o 16
v 1 2 1 1 1 1
c Found a better solution
o 1
v 1 4 7 8 3 4
s OPTIMUM FOUND
```

7.4   Diagnostics

A *diagnostic* is a (name,value) pair which describes the work carried out by the solver. They have to be written to stdout as a 'd ' line. 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 the its value. The following diagnostics are predefined:

CHECKS: The total number of *consistency checks* which have been carried out.

SAC_CHECKS: The total number of *singleton arc consistency checks* which have been carried out.

ASSIGNMENTS: The total number of *assignments* in the search tree.

Contestants wishing to record other diagnostics than the ones listed before should inform Marc van Dongen (dongen⟨at⟩cs.ucc.ie) about the names and nature of the diagnostics.

8   Entering the Competition

For **CSP**, **Max-CSP** and **WCSP**, the contestants can enter the competition with one or two solvers per solver categories (complete/incomplete). 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 25 instances (that

can be kept hidden) which will be guaranteed to be used for the competition. They will also have to submit a position paper (at least 3 pages) in a second stage.

We expect that contestants propose solvers that recognise the XML format XCSP 2.1 (either natively or by embedding a conversion procedure). Contestants whose solver cannot be interfaced to read the XML format must get in touch with the organisers.

The deadline for submitting both benchmarks and solvers is May 30, 2009. Submission of solvers will be available online in May 2009 at `http://www.cril.univ-artois.fr/CPAI09/`. Benchmarks may also be found at `http://www.cril.univ-artois.fr/CPAI09/`.

9 Ranking

For each problem (**CSP**, **Max-CSP** and **WCSP**) and each combination of instance and solver category, there will be a ranking of solvers. The categories are described in Section 4. The main criteria for ranking the solvers are as follows.

CSP: Solvers claiming incorrect results in a given category will be disqualified from this category. Of the remaining solvers, the solver solving the most problems will be declared the winner. Ties will be broken by considering the minimum total solution time.

Max-CSP/WCSP: Solvers claiming incorrect results in a given category will be disqualified from this category. The remaining solvers will be ranked according to the following:

Incomplete solvers will be ranked in increasing order of their average relative error (i.e. the average normalised difference between the cost of solutions found by the solver and the cost of the best known solutions).

Complete solvers will be ranked in decreasing order of the number of instances for which a solution has been proved to be OPTIMUM. To break ties, we will rank the solvers in increasing order of their average relative errors.

10 Competition committees

10.1 Organising committee

The following people are in charge of running the competition. They can be reached at cspcomp⟨at⟩cril.univ-artois.fr.

- Marc VAN DONGEN, dongen⟨at⟩cs.ucc.ie,
  University College Cork, Ireland.

- Christophe LECOUTRE, lecoutre⟨at⟩cril.fr,
  Centre de Recherches en Informatique de Lens, Université d'Artois, France.

- Olivier ROUSSEL, olivier.roussel⟨at⟩cril.univ-artois.fr,
  Centre de Recherches en Informatique de Lens, Université d'Artois, France.

10.2   Judges

Three judges 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 assigned to an independent committee which is nominated by the judges. The judges can be reached at cspcomp-jury⟨at⟩cril.univ-artois.fr. They are:

- Helmut SIMONIS, h.simonis⟨at⟩4c.ucc.ie,
  Cork Constraint Computation Centre (4C), Ireland.

- George KATSIRELOS, george.katsirelos⟨at⟩nicta.com.au,
  NICTA, Sydney, Australia.

- Matt STREETER, matts⟨at⟩cs.cmu.edu,
  Google, USA.

10.3   Working Group

A working group in charge of generating/collecting instances and developing tools has been formed. It can be reached at cspcomp-wg⟨at⟩cril.univ-artois.fr. The current members of the working group are as follows.

- Emmanuel HEBRARD, e.hebrard⟨at⟩4c.ucc.ie,
  Cork Constraint Computation Centre (4C), Ireland

For sake of transparency, working group members who are participating in the competition must make public any instance which they have developed/generated as a member of the working group at least one month before the final registration deadline.

The organisers kindly invite anybody who is interested in joining the working group to contact Emmanual Hebrard.