Solver Requirements for the Competition of Pseudo-Boolean Solvers

Olivier ROUSSEL

roussel@cril.fr

Version of this document

The version number and the date of this document (as recorded by the versioning system) are given below. They let you identify quickly if you have the most recent version of this document.

```
Version: $Rev: 4559 $
Last modification: $Date: 2024-04-09 19:31:21 +0200 (Tue, 09 Apr 2024) $
```

1 Introduction

This document details the output format that a solver must respect in order to enter the pseudo-Boolean competition. Section 2 details how the solver must communicate its results. Section 3 explains how solvers are ranked and what happens when a solver is found to be incorrect. At last, Section 4 details the requirements for submitting a solver.

2 Output Format

Solvers must print 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 may be used to manually check some results.

2.1 Messages

With the exception of the ”o ” line, there is no specific order in the solvers output lines. However, all lines, according to its first char, must belong to one of the four following categories:

- **comments (”c ” lines)**

These lines start by the two 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 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, outputing 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.

- **value of the objective function (for PBO), or current cost (for WBO) ("o " lines)**

  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 should be printed only for optimisation instances (PBO, WBO). They will be ignored for PBS instances.

  Whenever the solver finds a solution with a better value of the objective function (PBO) or of the current cost (WBO), it is asked to print an "o " line with the current value of the objective function/cost. Therefore, an "o " line must contain the lower case o followed by a space and then by an integer which represents the better value of the objective function/cost. The integer output on this line must be the value of the objective function/cost as found in the instance file. "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.

  Example:

  The instance file contains an objective function `min: 1 x1 +1 x2 -1 x3`

  Let f be this objective function found in the file.

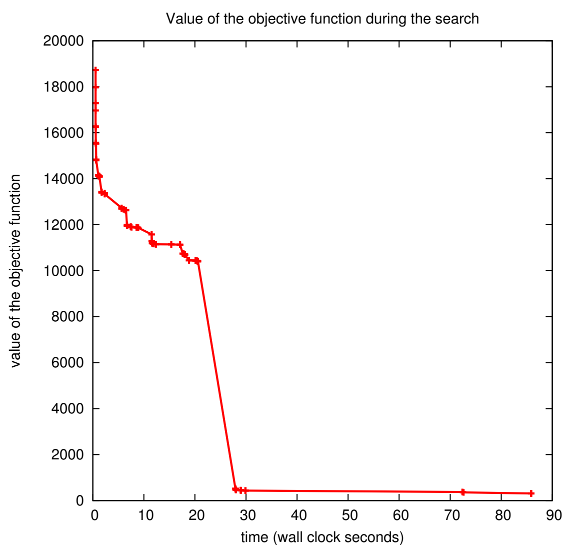
  The solver chooses to rewrite this function as $f' = x_1 + x_2 + not(x_3)$ to get only positive weights. It must remember that $f = f' - 1$ (since $-x = not(x) - 1$). When it finds a solution x_1=true, x_2=true, x_3=false, it must output "o 2" (f' has value 3 with this assignment but f has value 2). If x_1=false, x_2=false and x_3=true is a solution, the solver may successively output

```
o 2
o 1
o -1
s OPTIMUM FOUND
v -x1 -x2 x3
```

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 analyse 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 beginning 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 No problem reductions applied in OPT. instance.
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 -x1 -x2 -x3 x4 -x5 -x6 -x7 -x8 -x9 -x10 -x11 -x12 -x13 -x14 -x15
84.61    v -x16 -x17 -x18 -x19 -x20 -x21 -x22 -x23 -x24 -x25 -x26 -x27 -x28
84.61    c Total time: 84.478 s
```

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



- **solution ("s " lines)**

  This line starts by the two characters: lower case s followed by a space (ASCII code 32).

  Only one such line is allowed.

  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 input file contains a feature that the solver does not support. This answer may only be given at parse time, that is, before the solver starts exploring the solutions space. As an example, a solver which only understand linear constraint should print this line when it reads a non-linear constraint. Or a solver that finds an integer in the input file that exceeds its internal limit should report this answer at parse time.

    After the solver has parsed the input file, it must not print `s UNSUPPORTED`. It may use the `s UNKNOWN` answer instead.

  – `s SATISFIABLE`
    This line indicates that the solver has found a model of the formula, and in such a case, a "v " line is mandatory.

    For decision problems, this line must be printed when the solver has found a solution.

    For optimization problems, this line must be in the output when the solver has found a solution but it can not prove that this solution gives the best value of the objective function (PBO) or of the cost (WBO).

  - `s OPTIMUM FOUND`
    This line must be printed when the solver has found a model and it can prove that no other solution is better.

    For PBO instances, this means that no other solution will give a better value of the objective function than the one obtained with this model. Let v be the value of the objective obtained with the valuation output by the solver. Giving this result is a commitment that the formula extended with the constraint "objective<v" is unsatisfiable.

    For WBO instances, this answer means that no other interpretation has a smaller cost.

    This answer must not be used for PBS instances.
  - `s UNSATISFIABLE`
    This line must be output when the solver can prove that the formula 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 formula.

  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 *solution* line, or if the solution line is misspelled, then UNKNOWN will be assumed.

- **values ("v " lines)**

  This line starts by the two characters: lower case v followed by a space (ASCII code 32).

  More than one "v " line is allowed but the evaluation environment will act as if their content was merged.

  It is mandatory when the instance is satisfiable.

  If the solver finds a solution (it outputs "s SATISFIABLE" or "s OPTIMUM FOUND"), it must provide a solution. For PBS or PBO, this solution is a model (or an implicant) of the instance that will be used to check the correctness of the answer, i.e., it must provide a list of non-contradictory literals which, when interpreted to true, makes every constraint of the input formula true. For WBO, the solution is just an interpretation that satisfies each hard constraint and minimizes the cost of unsatisfied soft constraints. When optimization is considered, this set of literals should provide an interpretation such that the value of the objective funtion/cost corresponds to the best one that the solver was able to find. The negation of a literal is denoted by a minus sign immediately followed by the identifier of the variable. The solution line MUST define the value of EACH VARIABLE. The order of literals does not matter. Arbitrary white space characters, including ordinary white spaces, newline and tabulation characters, are allowed between the literals, as long as each line containing the literals is a *values* line, i.e. it begins with the two characters ”v ”.

The *values* lines should only appear with SATISFIABLE instance (including instances for which an OPTIMUM was FOUND).

Values lines must be terminated by a Line Feed character (the usual Unix line terminator '\n'. 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 print a complete solution.

For instance, the following outputs are valid for the instances given in example:

```
mycomputer: $ ./mysolver myinstance-sat
c mysolver 6.55957 starting with TIMEOUT fixed to 1000s
c Trying to guess a solution...
s SATISFIABLE
v x1 x2 -x5 x4 -x3
c Done (mycputime is 234s).

mycomputer: $ ./mysolver myinstance-unsat
c mysolver 6.55957 starting with TIMEOUT fixed to 1000s
c Trying to guess a solution...
c Contradiction found!
s UNSATISFIABLE
c Done (mycputime is 2s).
```

- **diagnostic (”d ” lines)**

  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.

  A diagnostic is simply an information that should be recorded because it is considered relevant. It is collected by the competition environment as a pair (keyword, value) and will be available later for further analysis.

  An example of diagnostic is the number of conflicts, which is relevant for a CDCL solver. A solver could print the line ”d CONFLICTS 1234” at the end of its execution to record that 1234 conflicts were encountered.

  If a diagnostic is encountered several times (same keyword), the keyword will be associated to an array of values.

  There is a limit on the number of diagnostics that can be recorded. If the solver is too verbose, some diagnostics may be lost.

3 Ranking of solvers

Basically, solvers will be ranked on the number of definitive answers[1] they give in each category. Ties are broken on the cumulated CPU/wall-clock time to give these answers.

[1]Definitive answers in the DEC categories are SATISFIABLE and UNSATISFIABLE, definitive answers in the OPT and SOFT categories are OPTIMUM FOUND and UNSATISFIABLE

Other ranking schemes may be introduced to help identify remarkable features.

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 PBS and PBO, it outputs SATISFIABLE or OPTIMUM FOUND, but provides an assignment which does not satisfy each 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 interpretation with a better value of the objective function/cost that the one obtained from the interpretation found.
- It generates an UNSAT/OPT certificate that is invalid.

When a solver provides even one single wrong answer in a given category of benchmarks, the solver’s results in that category cannot be trusted. Such a solver will not appear in the final ranking, but its results will still be published on the competition web site. A solver cannot be withdrawn from the competition once CPU time has been consumed to evaluate its performances.

A solver which ends without giving any solution, or just crashes for some reason (internal bugs...), is simply considered as giving an UNKNOWN result. It is bugged, but not incorrect.

4 Requirements for submitting a solver

4.1 Categories

Because not all solvers are able to solve every kind of instances, you will be asked when you register your solver to indicate which category of benchmarks it is able to handle. Up to 16 categories can be defined by the ability of the solver to:

- deal with PBS, PBO or WBO instances,
- deal with linear or non linear constraints,
- produce certificates of unsatisfiability/optimality.

Four categories are defined based on the kind of problem to solve:

- Category DEC (decision problem, pseudo-Boolean satisfaction (PBS))
  Benchmarks in this category contain neither an objective function nor soft constraints. The solver is expected to answer SATISFIABLE or UNSATISFIABLE. All solvers should register in this category. This category was called SAT/UNSAT in previous evaluations.

- Category OPT (pseudo-Boolean optimisation problem (PBO))
  Benchmarks in this category contain an objective function which should be minimized (and hence contain no soft constraint). Complete solvers entering this category must be able to find the best solution and give an OPTIMUM FOUND answer.

- Category SOFT (weighted boolean optimisation problem (WBO))
  Benchmarks in this category contain only soft constraints. Complete solvers entering this category must be able to find the best solution and give an OPTIMUM FOUND answer.

- Category PARTIAL (weighted boolean optimisation problem (WBO))
  Benchmarks in this category contain both soft and hard constraints. Complete solvers entering this category must be able to find the best solution and give an OPTIMUM FOUND answer.

Two categories are defined based on the kind of contraints contained in the instance:

- Category ”linear constraints” (LIN)
  All contraints in the instance are linear pseudo-Boolean constraints. All solvers should register in this category.

- Category ”non linear constraints” (NLC)
  At least one constraint (or the objective function) is non linear, meaning that it contains at least one product of Boolean variables. A non linear constraint or objective function can always be linearized by introducing new variables.

Two categories are defined based on to generate a certifcate of unsatisfiability/optimality: the instance:

- Category ”no certificate” (NOCERT)
  The solver will not generate a certificate of unsatisfiability/optimality. All solvers should register in this category.

- Category ”certificate” (CERT)
  The solver will generate a certificate of unsatisfiability/optimality in the VeriPB format [BGMN23, GN21, Goc22]. The solver has to store its proof in a file. The name of this file will be given to the solver on the command line (see the keyword PROOFFILE in Section 4.3).

4.2 Complete and incomplete solvers

Complete solvers are solvers which can always decide if the formula is satisfiable or not, provided they are given enough time and memory. Incomplete solvers are able to give some answers (e.g. SATISFIABLE) but not all. They may loop endlessly in a number of cases. Local search algorithms are examples of incomplete solvers. There is a high probability that they find a solution if the instance is satisfiable, but they will not be able to prove unsatisfiability.

Both kinds of solvers are welcome in the competition. Submitters will have to indicate if their solver is complete or incomplete on the submission form.

4.2.1 Complete solvers

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

4.2.2 Incomplete solvers

Incomplete solvers are definitely welcome in the competition. Despite the fact that they will never answer UNSATISFIABLE or OPTIMUM FOUND, incomplete solvers can be registered in each of the DEC, OPT and SOFT categories.

In the DEC category, 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.

In the two optimisation categories, 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 informations in these categories, an incomplete solver must fulfill two requirements:

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

2. it MUST output "o " lines whenever it finds a better solution so that, even if the solver always timeout, 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.

4.3 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 unauthorized use of the system (network connections, file creation outside the allowed directory, among others).

Solvers will be run as 64 bits applications. Authors are invited to submit solvers in source form, but this is not mandatory. If you submit an executable, you are required to provide us with an ELF executable (preferably statically linked).

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).

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 informations by the evaluation environment.

- BENCHNAME will be replaced by the name of the file containing the 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).

- PROOFFILE will be replaced by the name of the file that should contain the proof of unsatisfiability that may be generated by the solver. This file will not exist by default and should be created by the solver.
- 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/file.opb 1720968
/solver10/mysolver --mem-limit=900 --time-limit=1200 \
                   --tmpdir=/tmp/job12345 /tmp/file.opb
java -jar /solver10/mysolver.jar -c /solver10/mysolver.conf /tmp/file.opb
```

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.

4.4 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.

References

[BGMN23] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström, *Certified dominance and symmetry breaking for combinatorial optimisation*, Journal of Artificial Intelligence Research **77** (2023), 1539–1589, Preliminary version in *AAAI '22*.

[GN21] Stephan Gocht and Jakob Nordström, *Certifying parity reasoning efficiently using pseudo-Boolean proofs*, Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21), February 2021, pp. 3768–3777.

[Goc22] Stephan Gocht, *Certifying correctness for combinatorial algorithms by using pseudo-Boolean reasoning*, Ph.D. thesis, Lund University, Lund, Sweden, June 2022, Available at `https://portal.research.lu.se/en/publications/certifying-correctness-for-combinatorial-algorithms-by-using-pseu`.