ACE
A Generic Constraint Solver

Christophe Lecoutre
University of Artois
CRIL CNRS, UMR 8188
France

lecoutre@cril.fr

Version 2.0 – December 15, 2021

Abstract

ACE (AbsCon Essence) is a constraint solver (written in Java) that can solve constraint satisfaction/optimization problems. More specifically, you can solve instances of CSP (Constraint Satisfaction Problem) and COP (Constraint Optimization Problem) frameworks. ACE allows us to use integer variables (with finite domains) and popular constraints, those defined in XCSP^3-core. This document gives some general information about the structure of the code of ACE.

ACE (AbsCon Essence) is an open-source constraint solver, developed in Java. ACE focuses on:

- integer variables, including 0/1 (Boolean) variables,
- state-of-the-art table constraints, including ordinary, starred, and hybrid table constraints,
- popular global constraints (AllDifferent, Count, Element, Cardinality, Cumulative, etc.),
- search heuristics (wdeg, impact, last-conflict, BIVS, solution-saving, ...),
- mono-criterion optimization

ACE is derived from the constraint solver AbsCon that has been used as a research platform in our team at CRIL during many years. Many ideas and algorithms have been discarded from AbsCon, so as to get a constraint solver of reasonable size and understanding.

As we shall see in this document, for running the solver on an instance (XCSP^3 file), you have to execute:

```
java ace <xcsp3_file> [options]
```

with:

- <xcsp3_file>: an XCSP^3 file representing a CSP or COP instance
- [options]: possible options to be used when running the solver

For generating XCSP^3 instances, just write and compile models with the Python library PyCSP^3. See https://github.com/xcsp3team/pycsp3

Licence. ACE is licensed under the MIT License

Code. ACE code is available

- on Github: https://github.com/xcsp3team/ace

The code of ACE is structured in 13 packages.

1 General Packages

1.1 Package `main`

The package `main` contains two classes:

- `Head`, this is the class of the main object (head) in charge of solving a CSP or COP instance. Such an object solicits a solver in order to solve a problem.
- `HeadExtraction`, this is the class of the main object in charge of extracting an unsatisfiable core of constraints.

To run the solver on the problem instance `airTraffic.xml`, we can execute:

```
java main.Head airTraffic.xml
```

A shortcut exists, as a class is present in the default package:

```
java ace airTraffic.xml
```

To extract an unsatisfiable core of constraints from the (unsatisfiable) instance `Rlfap-scen-11-f08.xml`, we execute:

```
java main.HeadExtraction Rlfap-scen-11-f08.xml
```

1.2 Package `dashboard`

In the package `dashboard`, we find the following three classes:

- `Control`, which allows us to manage all options concerning the problem (typically, the way to represent it) and the solver (typically, the way to conduct search)
- `Input`, which allows us to handle arguments given by the user on the command line. These arguments may concern the problem to solve or more generally the solving process (i.e., options to choose like for example which search heuristics to use)
- `output`, whose role is to output some data/information concerning the solving process of problem instances. These data are collected by means of an XML document that may be saved. A part of the data are also directly displayed on the standard output.

The class `Control` is rather central, as it contains many important options. The user can specify the value of these options on the command line. As an example, in an intern class (representing an option group) of the class `Control`, we find:

```
long timeout = addL("timeout", "t", PLUS_INFINITY,
  "The limit in milliseconds before stopping; seconds can be indicated as in -t=10s");
```

The second parameter of the method `addL` is the shortcut to employ on the command line. So, to run the solver on the problem instance `airTraffic.xml` during at most 10 seconds, we need to execute:

```
java ace airTraffic.xml -t=10s
```

All options are briefly introduced in the appendix of this document.

1.3 Package `interfaces`

It is important to understand the role of the Java interfaces in the code of ACE.

1.3.1 Tags

Some interfaces are used as tags (i.e., they do not contain any piece of code):

- `TagSymmetric`, tag for indicating that a constraint is completely symmetric
- `TagNotSymmetric`, tag for indicating that a constraint is not symmetric at all
- `TagCallCompleteFiltering`, tag for indicating that a constraint can produce full filtering at each call (not only around the last touched variable)
- `TagAC`, tag for indicating that a constraint guarantees enforcing (G)AC
- `TagNotAC`, tag for indicating that a constraint does not guarantee enforcing (G)AC
- `TagNegative`, tag for indicating that a table constraint is negative (i.e., contains conflicts)
- `TagPositive`, tag for indicating that a table constraint is positive (i.e., contains supports)
- `TagStarredCompatible`, tag for indicating that a constraint may contain starred elements (i.e., may contain *)
- `TagMaximize`, tag for indicating that an object (e.g., an heuristic) aims at maximizing an expression (variable, sum, maximum, etc.)

1.3.2 Observers

Some interfaces are used for observing.

- `ObserverOnConstruction`, for observing the construction of the main objects (problem and solver). Methods are:
  - `beforeAnyConstruction`, called before the main objects (problem and solver) are started to be built
  - `afterProblemConstruction`, called when the construction of the problem is finished
  - `afterSolverConstruction`, called when the construction of the solver is finished
- `ObserverOnSolving`, for observing the main steps of solving (preprocessing and search). Methods are:
  - `beforeSolving`, called before solving (preprocessing followed by search) is started
  - `beforePreprocessing`, called before preprocessing is started
  - `afterPreprocessing`, called after preprocessing is finished
  - `beforeSearch`, called before (backtrack) search is started
  - `afterSearch`, called after (backtrack) search is finished
  - `afterSolving`, called after solving (preprocessing followed by search) is finished
- `ObserverOnRuns`, for observing successive runs (possibly, only one run if restarting is deactivated) performed by the solver. Methods are:
  - `beforeRun`, called before the next run is started
  - `afterRun`, called after the current run is finished
- `ObserverOnBacktracks`, for observing backtracks performed by the solver. Methods and subinterfaces are:

- `restoreBefore(int depthBeforeBacktrack)`, called when a restoration is required due to a backtrack coming from the specified depth
- interface `ObserverOnBacktracksSystematic`, for observing backtracks performed by the solver. Used for observers that systematically require restoration
- interface `ObserveronBacktracksUnsystematic`, for observing backtracks performed by the solver. Used for observers that does not systematically require restoration.

◦ `ObserverOnDecisions`, for observing decisions taken by the solver. Methods are:

- `beforePositiveDecision(Variable x, int a)`, called when a positive decision (variable assignment x=a) is going to be taken
- `beforeNegativeDecision(Variable x, int a)`, called when a negative decision (value refutation x !=a) is going to be taken

◦ `ObserverOnAssignments`, for observing assignments taken by the solver. Methods are:

- `afterAssignment(Variable x, int a)`, called after the variable assignment x=a has been taken
- `afterUnassignment(Variable x)`, called after the variable x has been unassigned (due to backtrack)

◦ `ObserverOnRemovals`, for observing reductions of domains (typically, when filtering) by the solver. Methods are:

- `afterRemoval(Variable x, int a)`, called when the index a (of a value) for the domain of x has been removed
- `afterRemovals(Variable x, int nRemovals)`, called when the domain of the variable x has been reduced; the number of deleted values is specified

◦ `ObserverOnConflicts`, for observing conflicts encountered during search by the solver. Methods are:

- `whenWipeout(Constraint c, Variable x)`, Called when the domain of the specified variable has become empty (so-called domain wipeout) when filtering with the specified constraint
- `whenBacktrack`, called when the solver is about to backtrack

1.3.3 Specific Filtering

There is an important interface that is used for specifying that a constraint has its own filtering algorithm (propagator).

```
public interface SpecificPropagator {

  /**
  * Runs the propagator (specific filtering algorithm) attached to the constraint
  * implementing this interface, and returns false if an inconsistency is detected.
  * We know that the specified variable has been picked from the propagation queue,
  * and has been subject to a recent reduction of its domain.
  *
  * @param evt
  *            a variable whose domain has been reduced
  * @return false if an inconsistency is detected
  */
  boolean runPropagator(Variable evt);
}
```

1.4 Packages `sets` and `utility`

There are three main implementations of sets (which are important structures) in ACE, which can be found in the package `sets`:

- `SetDense`; a dense set is basically composed of an array (of integers) and a limit: at any time, it contains the elements (typically, indexes of values) in the array at indexes ranging from 0 to the limit (included).
- `SetSparse`; a sparse set is basically composed of two arrays (of integers) and a limit: at any time, it contains the elements (typically, indexes of values) in the array 'dense' at indexes ranging from 0 to the limit (included). The presence of elements can be checked with the array 'sparse'. Sparse sets are always simple, meaning that values in 'dense' are necessarily indexes 0, 1, 2, ... Besides, we have `dense[sparse[value]] = value`.
- `SetLinked`; a linked set is an interface that allows us to represent a list of elements perceived as indexes, i.e., elements whose values range from 0 to a specified capacity -1. For instance, if the initial size (capacity) of the object is 10, then the list of indexes/elements is 0, 1, 2... , 9. One can remove indexes of the list, one can iterate, in a forward or backward way, the currently present indexes, and one can iterate over deleted indexes. Each deleted index has an associated level. This kind of interface is notably used for managing the indexes of values of variable domains.

There are several subclasses:

- `SetDenseReversible`; a dense set that can be handled at different levels (of search)
- `SetSparseReversible`; a sparse set that can be handled at different levels (of search)
- `SetLinkedBinary` and `SetLinkedFinite`; for sets containing only two elements (indexes 0 and 1), and ordered sets containing a finite number of elements (indexes), respectively.

In the package `utility`, there are several classes with utility methods:

- `Bit`, containing methods on bit vectors
- `Kit`, containing various useful methods
- `Reflector`, containing methods for performing operations based on reflection
- `Stopwatch`, containing methods for measuring the time taken by some operations, when solving a problem

2 Packages concerning the Problem Representation

2.1 Package `problem`

The package `problem` contains the main class `Problem`, as well as some auxiliary classes. The main classes of the package are:

- `Problem`, the class for representing the problem instance (constraint network)
- `Features`, the class that is useful for storing various information (features such as sizes of domains, types of constraints, etc.) about the problem instance, and ways of displaying it
- `Reinforcer`, the class containing inner classes that are useful to reinforce a constraint network by adding either redundant constraints or symmetry-breaking constraints
- `XCSP3`, the class that allows us to load instances in XCSP3 format. This class is the interface part while the class `Problem` is the implementation part. This separation is due to historical reasons (from the API JvCSP3), but could be removed in the future so as to simplify code.

2.2 Package `variables`

The package `variables` contains classes for defining both variables and domains. It also contains a class that is useful to iterate over the tuples of the Cartesian product of specified domains. The main classes of the package are:

- `Variable`, this is the abstract root class for any variable involved in a problem, with currently two inner subclasses: `VariableInteger` and `VariableSymbolic`. A domain is attached to a variable and corresponds to the (finite) set of values which can be assigned to it. When a value is assigned to a variable, the domain of this variable is reduced to this value. When a solver tries to assign a value to a variable, it uses a value ordering heuristic in order to determine which value must be tried first. A variable can of course occur in different constraints of the problem to which it is attached.
- `Domain`, this is the abstract root class for the domain of any variable. A domain is composed of a set of integer values. The domain is initially full, but typically reduced when logically reasoning (with constraints). When handling a domain, to simplify programming, one usually iterates over the indexes of the values; if the domains contains d values, the indexes then range from 0 to $d-1$. For instance, if the domain is the set of values $\{1, 4, 5\}$, their indexes are respectively $\{0, 1, 2\}$. The correspondence between indexes of values and values is given by the methods `toIdx` and `toVal`. Subclasses of `Domain` are:
  - `DomainBinary`, the class for binary domains typically containing 0 and 1 only.
  - `DomainRange`, the class for domains composed of a list of integers included between two (integer) bounds.
  - `DomainValues`, the class for domains composed of a list of integers that are not necessarily contiguous. Note that the values are sorted.

  Note that `DomainRange` and `DomainValues` are two inner subclasses of `DomainFinite`.
- `TupleIterator`, the class that allows us to iterate over the Cartesian product of the domains of a sequence of variables. For example, it allows us to find the first valid tuple, or the next tuple that follows a specified (or last recorded) one. Each constraint is equipped with such an object. It is important to note that iterations are performed with indexes of values, and not directly values.

2.3 Package `constraints`

The package `constraints` contains many classes representing various types of constraints. Constraints can be defined:

- extensionally, by means of tables or diagrams
- intensionally, by means of general Boolean tree expressions, which often correspond to classical forms called primitives
- from a well-known template (so-called global constraints).

Consequently, the more abstract classes from the package are:

- `Constraint`, the root class of any constraint. Let us recall that a constraint is attached to a problem, involves a subset of variables of the problem, and allows us to reason so as to filter the search space (i.e., the domains of the variables).
- `ConstraintExtension`, the root class for representing extension constraints, also called table constraints. Two direct subclasses are `ExtensionGeneric` (for implementing AC filtering à la AC3^{rm}) and `ExtensionSpecific` (for implementing specific propagators). Algorithms for filtering extension constraints can be found in the subpackage `extension`:

  - STR1, STR2, and STR3
  - CT
  - CMDD
  - ...

- `ConstraintIntension`, the root class for representing intension constraints, which are constraints whose semantics is given by a Boolean expression tree involving variables. Most of the times, primitives are used instead of this general form. Algorithms for filtering primitive and reified constraints can be found in the subpackage `intension`.
- `ConstraintGlobal`, the root class for representing global constraints, which are essentially constraints with a specific form of filtering (propagator). Algorithms for filtering global constraints can be found in the subpackage `global`.

3 Packages concerning the Solving Process

3.1 Package `solver`

The package `solver` contains the main class `Solver`, as well as some auxiliary classes that are useful for managing future variables (i.e., unassigned variables), decisions, solutions, restarts, last conflicts and statistics. The main classes of the package are:

- `Solver`, the class of the main object used to solve a problem.
- `FutureVariables`, the class of the object that allows us to manage past and future variables.
- `Decisions`, the class of the object that allows us to store the set of decisions taken by the solver during search. Each decision is currently encoded under the form of an int. However, it is planned in the future to represent each decision with a long instead of an int.
- `Solutions`, the class of the object used to record and display solutions (and bounds, etc.).
- `Restarter`, the root class of the object used for manage restarts (successive runs restarting from the root node).
- `LastConflicts`, the class of the object implementing last-conflict reasoning (lc). See "Reasoning from last conflict(s) in constraint programming", Artif. Intell. 173(18): 1592-1614 (2009) by C. Lecoutre, L. Sais, S. Tabary, and V. Vidal.
- `Statistics`, the class of the object that allows us to gather all statistics (as e.g., the number of backtracks) of the solver.

3.2 Package `propagation`

In the package `propagation`, we can find classes that implement different ways of conducting the filtering process (or constraint propagation), all inheriting from the root class `Propagation`. For example, among them, we find those corresponding to the consistencies AC (Generalized Arc Consistency), SAC (Singleton Arc Consistency) and GIC (Global Inverse Consistency). For simplicity, propagation and consistency concepts are not distinguished. This is why some subclasses are given the name of consistencies.

Among subclasses, we find:

- `Backward`, which is the root class for backward propagation. Such form of propagation corresponds to a retrospective approach that deals with assigned variables, meaning that the domains of the unassigned variables are never modified. Subclasses are `BT`, the basic backtracking algorithm, and `GT`, the "generate and test" algorithm.

- `Forward`, which is the root class for forward propagation. Such form of propagation corresponds to a prospective approach that deals with unassigned variables, meaning that the domains of the unassigned variables can be filtered. Subclasses are:
  - `FC`, the class for Forward Checking (FC)
  - `AC`, the class for (Generalized) Arc Consistency (AC). Such a propagation object solicits every constraint propagator (i.e., filtering algorithm attached to a constraint) until a fixed point is reached (contrary to FC). Note that it is only when every propagator ensures AC that AC is really totally enforced on the full constraint network. Let us recall that AC is the maximal level of possible filtering when constraints are treated independently. Note that, for simplicity, we use the acronym AC (and not GAC) whatever is the arity of the constraints.
  - `GIC`, the class for Global Inverse Consistency (GIC). This kind of consistency is very strong as once enforced, it guarantees that each literal (x,a) belongs to at least one solution. It can only be executed on some special problems. For example, see "Computing and restoring global inverse consistency in interactive constraint satisfaction", Artif. Intell. 241: 153-169 (2016) by C. Bessiere, H. Fargier, and C. Lecoutre.
  - `SAC`, the class for Singleton Arc Consistency (AC). Some information about SAC and algorithms enforcing it can be found for example in: "Efficient algorithms for singleton arc consistency", Constraints An Int. J. 16(1): 25-53 (2011), by C. Bessiere, S. Cardon, R. Debruyne, and C. Lecoutre.

3.3 Package `learning`

In the package `learning`, we can find classes related to *constraint learning*. Some of them are about nogoods, and some are about so-called Inconsistent Partial States (IPSs).

Remark 1 *Currently, for historic and maintenance reasons, IPS learning is deactivated. We plan, in the near future, to reactivate it.*

Entities that can be learned during search are from classes:

- `Nogood`; strictly speaking, an object of this class is used as if it was a nogood constraint, i.e. a disjunction of negative decisions that must be enforced (to be true). However, nogoods are handled apart during constraint propagation.
- `Ips`; an IPS is an Inconsistent Partial State. It is composed of membership decisions and is equivalent to generalized nogoods.

To identify and reason with such entities, we have the classes:

- `NogoodReasoner`, which allows us to record and reason with nogoods.
- `IPsReasoner`, which allows us to record and reason with inconsistent partial states (IPSs). It is possible to reason in term of equivalence or dominance.

Currently, learning is limited to *nogood recording from restarts*.

3.4 Package `heuristics`

3.4.1 Variable Ordering Heuristic

Variable ordering heuristics can be found in the package `heuristics`, implemented by Java classes inheriting from `HeuristicVariables`. Let us suppose that we want to implement the classical dynamic variable ordering heuristic `dom` that selects the variable with the smallest domain size. We have to write a class inheriting from `HeuristicVariablesDynamic` while specifying a body for the method `scoreOf`, as follows:

```
public class Dom extends HeuristicVariablesDynamic {
  public Dom(Solver solver, boolean anti) {
    super(solver, anti);
  }

  @Override
  public double scoreOf(Variable x) {
    return x.dom.size();
  }
}
```

In case the highest score must be selected (and not the smallest one, as for `dom`), the class must be tagged as follows: `implements TagMaximize`.

Once a new heuristic is available, we can select it by using the option `-varh`. For example, to run the solver on the problem instance `airTraffic.xml`, while using `dom` as variable ordering heuristic, we just have to execute:

```
java ace airTraffic.xml -varh=Dom
```

If for some reasons, we want to use the anti-heuristic, we execute:

```
java ace airTraffic.xml -varh=Dom -anti_varh
```

Among implemented heuristics, we find:

- `Rand`
- `Dom`
- `DDegOnDom`
- `Wdeg`
- `WdegOnDom`
- `Activity` and `Impact`, with basic implementations

For the constraint-weighting heuristics `Wdeg` and `WdegOnDom`, four variants exist:

- VAR, a basic variant
- UNIT, classical weighting, as described in "Boosting Systematic Search by Weighting Constraints", ECAI 2004: 146-150, by F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais.
- CACD, refined weighting, as described in "Refining Constraint Weighting", ICTAI 2019: 71-77 by H. Wattez, C. Lecoutre, A. Paparrizou, and S. Tabary
- CHS, as described in "Conflict history based search for constraint satisfaction problem", SAC 2019: 1117-1122 by D. Habet, and C. Terrioux

By default, the variant CACD of `Wdeg` is selected. For specifying a variant, the option `-wt` must be used. For example, to run the solver on `airTraffic.xml`, while using the variant CHS of `WdegOnDom`, we need to execute:

When running the solver on a problem instance, here `airTraffic.xml`, while using default options, we just have to execute:

```
java ace airTraffic.xml -varh=WdegOnDom -wt=chs
```

Note that the case is insensitive for `-wt`.

3.4.2 Value Ordering Heuristic

Value ordering heuristics can be found in the package `heuristics`, implemented by Java classes inheriting from `HeuristicValues`. Let us suppose that we want to implement the value ordering heuristic `rand` that randomly selects a value in the domain of the selected variable. We have to write a class inheriting from `HeuristicValuesDirect` while specifying a body for the method `computeBestValueIndex`, as follows:

```
public class Rand extends HeuristicValuesDirect {
  public Rand(Variable x, boolean dummy) {
    super(x, dummy);
  }

  @Override
  public int computeBestValueIndex() {
    return x.dom.any();
  }
}
```

Once a new heuristic is available, we can select it by using the option `-valh`. For example, to run the solver on the problem instance `airTraffic.xml`, while using `rand` as value ordering heuristic, we just have to execute:

```
java ace airTraffic.xml -valh=Rand
```

Among implemented heuristics, we find:

- `Rand`
- `First` and `Last`
- `Robin` and `RunRobin`
- `Bivs`, as defined in "Making the First Solution Good!", ICTAI 2017: 1073-1077

3.5 Package `optimization`

The package `optimization` contains classes that are useful for managing optimization. The main classes of the package are:

- `Optimizer`, the abstract root class of the object used as a pilot for dealing with (mono-objective) optimization. Subclasses define various strategies to conduct search toward optimality:
  - `OptimizerDecreasing`, an optimization strategy based on decreasingly updating the maximal bound (assuming minimization) whenever a solution is found; this is related to branch and bound (and related to ramp-down strategy)
  - `OptimizerIncreasing`, an optimization strategy based on increasingly updating the minimal bound (assuming minimization); this is sometimes called iterative optimization (or ramp-up strategy)
  - `OptimizerDichotomic`, an optimization strategy based on a dichotomic reduction of the bounding interval
- `Optimizable`, the interface that any constraint must implement so as to represent an objective
- `ObjectiveVariable`, the abstract root class of the object to be used when the objective of the problem instance (constraint network) is given by a variable whose value must be minimized or maximized. Inner subclasses are `ObjVarLE` and `ObjVarGE`.

Acknowledgements

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

Appendix: List of Options

To get a possibly updated list of options, just execute:

```
java ace <xcsp3_file> [options]
```

These options are:

```
General
  -General/solLimit -s
    The limit on the number of found solutions before stopping; for no limit, use -s=all
    Default value is: -1
  -General/timeout -t
    The limit in milliseconds before stopping; seconds can be indicated as in -t=10s
    Default value is: 9223372036854775807
  -General/discardClasses -dc
    XCSP3 classes (tags) to be discarded (comma as separator)
    Default value is: "" (empty string)
  -General/trace -trace
    Displays a trace (with possible depth control as eg -trace=10-20
    Default value is: "" (empty string)
  -General/jsonLimit -jl
    The limit on the number of variables for displaying solutions in JSON
    Default value is: 1000
  -General/jsonAux -ja
    Take auxiliary variables when displaying solutions in JSON
    Default value is: false
  -General/jsonSave -js
    Save the first solution in a file whose name is this value
    Default value is: "" (empty string)
  -General/jsonQuotes -jq
    Surround keys with quotes when solutions are displayed on the standard output
    Default value is: false
  -General/xmlCompact -xc
    Compress values when displaying solutions in XML
    Default value is: true
  -General/xmlEachSolution -xe
    During search, display all found solutions in XML
    Default value is: false
  -General/noPrintColors -npc
    Don't use special color characters in the terminal
    Default value is: false
  -General/exceptionsVisible -ev
    Makes exceptions visible.
    Default value is: false
  -General/enableAnnotations -ea
    Enables annotations (currently, mainly concerns priority variables).
    Default value is: false
  -General/satisfactionLimit -satl
    Converting the objective into a constraint with this limit
    Default value is: 2147483647
  -General/seed -seed
    The seed that can be used for some random-based methods.
    Default value is: 0
  -General/verbose -v
    Verbosity level (value between -1 and 3)
```

```
    0 : only some global statistics are listed;
    1 : in addition, solutions are  shown;
    2 : in addition, additionnal information about the problem instance to be solved is given;
    3 : in addition, for each constraint, allowed or unallowed tuples are displayed.
    Default value is: 0

Problem
  -Problem/shareBits -shareBits
    Trying to save space by sharing bit vectors.
    Default value is: false
  -Problem/symmetryBreaking -sb
    Symmetry-breaking method (requires Saucy to be installed
    Default value is: NO
    Possible values: NO SB_LE SB_LEX

Variables
  -Variables/selection -sel
    Allows us to give a list of variable that will form the subproblem to be solved.
    This list is composed of a sequence of tokens separated by commas (no whitespace).
    Each token is a variable id a variable number or an interval of the form i..j with i and j integers.
    For example, -sel=2..10,31,-4 will denote the list 2 3 5 6 7 8 9 10 31.
    This is only valid for a XCSP instance.
    Default value is: "" (empty string)
  -Variables/instantiation -ins
    Allows us to give an instantiation of variables for the problem to be solved.
    This instantiation is given under the form vars:values.
    vars is a sequence of variable ids or numbers separated by commas (no whitespace).
    values is a sequence of integers (the values for variables) separated by commas (no whitespace).
    For example, -ins=x2,x4,x9:1,11,4 will denote the instantiation {x2=1,x4=11,x9=4}.
    Default value is: "" (empty string)
  -Variables/priority1 -pr1
    Allows us to give a list of variables that will become strict priority variables (to be used
    by the variable ordering heuristic). This list is composed of a sequence of strings (ids of variables)
    or integers (numbers of variables) separated by commas (no whitespace).
    For example, -pr1=2,8,1,10 will denote the list 2 8 1 10.
    Default value is: "" (empty string)
  -Variables/priority2 -pr2
    Allows us to give a list of variables that will become (non strict) priority variables.
    This list is composed of a sequence of tokens separated by commas (no whitespace).
    Each token is variable id, a variable number (integer) or an interval of the form i..j
    with i and j integers. For example, -pr2=2..10,31,-4 will denote the list 2 3 5 6 7 8 9 10 31.
    Default value is: "" (empty string)
  -Variables/reduceIsolated -riv
    Arbitrary keeping a single value in the domain of isolated variables
    Default value is: true

Constraints
  -Constraints/preserve1 -pc1
    Must we keep unary constraints (instead of filtering them straight)
    Default value is: true
  -Constraints/ignoreType -ignoreType
    Dicard all constraints of this type
    Default value is: "" (empty string)
  -Constraints/ignoreArity -ignoreArity
    Discard all constraints of this arity
    Default value is: -1
```

```
  -Constraints/positionsLb -poslb
    Minimal arity to build the array positions
    Default value is: 3
  -Constraints/positionsUb -posub
    Maximal number of variables to build the array positions
    Default value is: 10000

Optimization
  -Optimization/lb -lb
    Initial lower bound
    Default value is: -9223372036854775808
  -Optimization/ub -ub
    Initial upper bound
    Default value is: 9223372036854775807
  -Optimization/strategy -os
    Optimization strategy
    Default value is: DECREASING
    Possible values: INCREASING DECREASING DICHOTOMIC

Extension
  -Extension/positive -positive
    Algorithm for (non-binary) positive table constraints
    Default value is: CT
    Possible values: V VA STR1 STR2 STR3 STR1N STR2N CT CMDDO CMDDS
  -Extension/negative -negative
    Algorithm for (non-binary) negative table constraint
    Default value is: V
    Possible values: V VA STR1 STR2 STR3 STR1N STR2N CT CMDDO CMDDS
  -Extension/generic2 -generic2
    Must we use a generic filtering scheme for binary table constraints?
    Default value is: true
  -Extension/structureClass2 -sc2
    Structures to be used for binary table constraints
    Default value is: Bits
    Possible values: Matrix3D TableHybrid Tries Bits MDD Matrix2D Table
  -Extension/structureClass3 -sc3
    Structures to be used for ternary table constraints
    Default value is: Matrix$Matrix3D
    Possible values: Matrix3D TableHybrid Tries Bits MDD Matrix2D Table
  -Extension/arityLimitToPositive -alp
    Limit on arity for converting negative table constraints to positive
    Default value is: -1
  -Extension/arityLimitToNegative -aln
    Limit on arity for converting positive table constraints to negative
    Default value is: -1
  -Extension/variant -extv
    Variant to be used for some algorithms (e.g., VA or CMDD)
    Default value is: 0
  -Extension/decremental -extd
    Must we use a decremental mode for some algorithms (e.g., STR2, CT or CMDD)
    Default value is: true

Intension
  -Intension/decompose -di
    0: no decomposition, 1: conditional decomposition, 2: forced decompostion
    Default value is: 1
```

```
  -Intension/toExtension1 -ie1
    Must we convert unary intension constraints to extension?
    Default value is: true
  -Intension/arityLimitToExtension -ale
    Limit on arity for possibly converting to extension
    Default value is: 0
  -Intension/spaceLimitToExtension -sle
    Limit on space for possibly converting to extension
    Default value is: 20
  -Intension/recognizePrimitive2 -rp2
    Must we attempt to recognize binary primitives?
    Default value is: true
  -Intension/recognizePrimitive3 -rp3
    Must we attempt to recognize ternary primitives?
    Default value is: true
  -Intension/recognizeReifLogic -rlog
    Must we attempt to recognize logical reification forms?
    Default value is: true
  -Intension/recognizeExtremum -rext
    Must we attempt to recognize minimum/maximum constraints?
    Default value is: true
  -Intension/recognizeSum -rsum
    Must we attempt to recognize sum constraints?
    Default value is: true

Global
  -Global/allDifferent -g_ad
    Algorithm for AllDifferent
    Default value is: 0
  -Global/allDifferentExcept -g_ade
    Algorithm for AllDifferentExcept
    Default value is: 0
  -Global/distinctVectors -g_dv
    Algorithm for DistinctVectors
    Default value is: 0
  -Global/allEqual -g_ae
    Algorithm for AllEqual
    Default value is: 0
  -Global/notAllEqual -g_nae
    Algorithm for NotAllEqual
    Default value is: 0
  -Global/noOverlap -g_no
    Algorithm for NoOverlap
    Default value is: 0
  -Global/redundNoOverlap -r_no
    Must we post redundant constraints for NoOverlap?
    Default value is: true
  -Global/binpacking -g_bp
    Algorithm for BinPacking
    Default value is: 0
  -Global/viewForSum -vs
    Must we use views for Sum constraints, when possible?
    Default value is: false
  -Global/permutation -permutation
    Must we use permutation constraints for AllDifferent if possible? (may be faster)
    Default value is: false
```

```
  -Global/allDifferentNb -adn
    Number of possibly automatically inferred AllDifferent
    Default value is: 0
  -Global/allDifferentSize -ads
    Limit on the size of possibly automatically inferred AllDifferent
    Default value is: 5
  -Global/starred -starred
    When true, some global constraints are encoded by starred tables
    Default value is: false
  -Global/hybrid -hybrid
    When true, some global constraints are encoded by hybrid/smart tables
    Default value is: false

Propagation
  -Propagation/clazz -p
    Class to be used for propagation (for example, FC, AC or SAC3)
    Default value is: AC
    Possible values: GIC2 SAC SAC3 ESAC3 GT FC GIC3 BT GIC AC GIC4 SAC3p
  -Propagation/variant -pv
    Propagation Variant (only used for some consistencies)
    Default value is: 0
  -Propagation/useAuxiliaryQueues -uaq
    Must we use auxiliary queues of constraints?
    Default value is: false
  -Propagation/reviser -reviser
    Class to be used for performing revisions
    Default value is: Reviser$Reviser3
    Possible values: Reviser3 Reviser2 Reviser
  -Propagation/residues -res
    Must we use redidues (AC3rm)?
    Default value is: true
  -Propagation/bitResidues -bres
    Must we use bit resides (AC3bit+rm)?
    Default value is: true
  -Propagation/multidirectionality -mul
    Must we use multidirectionality
    Default value is: true
  -Propagation/futureLimit -fl
    AC not enforced when the number of future variables is greater than this value (if not -1)
    Default value is: -1
  -Propagation/spaceLimit -sl
    AC not enforced when size of the Cartesian product of domains is greater than 2
    to the power of this value (if not -1)
    Default value is: 20
  -Propagation/arityLimit -al
    AC not enforced if the arity is greater than this value
    Default value is: 2
  -Propagation/strongOnce -so
    Must we only apply the strong consistency (if chosen) before search?
    Default value is: false
  -Propagation/strongAC -sac
    Must we only apply the strong consistency (if chosen) when AC is effective?
    Default value is: false

Learning
  -Learning/nogood -ng
```

```
    Nogood recording technique (from restarts by default)
    Default value is: RST
    Possible values: NO RST RST_MIN RST_SYM
  -Learning/nogoodBaseLimit -ngbl
    The maximum number of nogoods that can be stored in the base
    Default value is: 200000
  -Learning/nogoodArityLimit -ngal
    The maximum arity of a nogood that can be recorded
    Default value is: 2147483647
  -Learning/ips -ips
    IPS extraction technique (currently, no such learning by default)
    Default value is: NO
    Possible values: NO EQUIVALENCE DOMINANCE
  -Learning/ipsOperators -ipso
    Reduction operators for IPSs; a sequence of 5 bits is used
    Default value is: 11011
  -Learning/ipsCompression -ipsc
    IPS Compression for equivalence reasoning
    Default value is: 0
  -Learning/ipsCompressionLimit -ipscl
    IPS Compression limit for equivalence reasoning
    Default value is: 300

Solving
  -Solving/clazz -s_class
    Class for solving (usually, Solver)
    Default value is: Solver
    Possible values: Solver
  -Solving/enablePrepro -prepro
    Must we perform preprocessing?
    Default value is: true
  -Solving/enableSearch -search
    Must we perform search?
    Default value is: true
  -Solving/branching -branching
    Branching scheme for search (binary or non-binary)
    Default value is: BIN
    Possible values: BIN NON

Restarts
  -Restarts/nRuns -r_n
    Maximal number of runs (restarts) to be performed
    Default value is: 2147483647
  -Restarts/cutoff -r_c
    Cutoff as a value of, e.g., the number of failed asignments before restarting
    Default value is: 10
  -Restarts/factor -r_f
    The geometric increasing factor when updating the cutoff
    Default value is: 1.1
  -Restarts/measure -r_m
    The metrics used for measuring and comparing with the cutoff
    Default value is: FAILED
    Possible values: FAILED WRONG BACKTRACK SOLUTION
  -Restarts/resetPeriod -r_rp
    Period, in term of number of restarts, for resetting restart data.
    Default value is: 90
```

```
  -Restarts/resetCoefficient -r_rc
    Coefficient used for increasing the cutoff, when restart data are reset
    Default value is: 2
  -Restarts/varhResetPeriod -r_vrp
    Description is missing...
    Default value is: 2147483647
  -Restarts/varhSolResetPeriod -r_vsrp
    Description is missing...
    Default value is: 30
  -Restarts/restartAfterSolution -ras
    Must we restart every time a solution is found?
    Default value is: false
  -Restarts/luby -luby
    Must we use a Luby series instead of a geometric one?
    Default value is: false

LNS
  -LNS/enabled -lns_e
    Must we activate LNS?
    Default value is: false
  -LNS/clazz -lns_h
    Class of the freezing heuristic
    Default value is: Restarter$RestarterLNS$HeuristicFreezing$Rand
    Possible values: Rand Impact
  -LNS/nFreeze -lns_n
    Number of variables to freeze when restarting
    Default value is: 0
  -LNS/pFreeze -lns_p
    Percentage of variables to freeze when restarting
    Default value is: 10

Revh
  -Revh/clazz -revh
    Class of the revision ordering heuristic
    Default value is: HeuristicRevisions$HeuristicRevisionsDynamic$Dom
    Possible values: Wdeg WdegOnDom Ddeg Dom DdegOnDom Rand First Lexico Last
  -Revh/anti -anti_revh
    Must we use the reverse of the natural heuristic order?
    Default value is: false

Varh
  -Varh/clazz -varh
    Class of the variable ordering heuristic
    Default value is: HeuristicVariablesDynamic$Wdeg
    Possible values: Lexico Dom Rand WdegOnDom Impact Memory Wdeg DdegOnDom Activity Ddeg Deg DomThenDeg
  -Varh/anti -anti_varh
    Must we use the reverse of the natural heuristic order?
    Default value is: false
  -Varh/lc -lc
    Value for lc (last conflict); 0 if not activated
    Default value is: 2
  -Varh/weighting -wt
    How to manage weights for wdeg variants
    Default value is: CACD
    Possible values: VAR UNIT CACD CHS
  -Varh/singleton -sing
```

```
    How to manage singleton variables during search
    Default value is: LAST
    Possible values: ANY FIRST LAST
  -Varh/connected -connected
    Must we select a variable necessarily connected to an already explicitly assigned one?
    Default value is: false
  -Varh/discardAux -da
    Must we not branch on auxiliary variables introduced by the solver?
    Default value is: false

Valh
  -Valh/clazz -valh
    Class of the value ordering heuristic
    Default value is: HeuristicValuesDirect$First
    Possible values: Rand Bivs2 Conflicts RunRobin Values Occurrences Robin Last Bivs First Median Failures
  -Valh/anti -anti_valh
    Must we use the reverse of the natural heuristic order?
    Default value is: false
  -Valh/runProgressSaving -rps
    Must we use run progress saving?
    Default value is: false
  -Valh/solutionSaving -sos
    Solution saving (0: disabled, 1: enabled, otherwise desactivation period
    Default value is: 1
  -Valh/warmStart -warm
    A starting instantiation (solution) to be used with solution saving
    Default value is: "" (empty string)
  -Valh/bivsFirst -bivs_f
    Must we stop BIVS at first found solution?
    Default value is: true
  -Valh/bivsOptimistic -bivs_o
    Must we use the optimistic BIVS mode?
    Default value is: true
  -Valh/bivsDistance -bivs_d
    0: only if in the objective constraint; 1: if at distance 0 or 1; 2: any variable
    Default value is: 2
  -Valh/bivsLimit -bivs_l
    BIVS applied only if the domain size is <= this value
    Default value is: 2147483647
  -Valh/optValHeuristic -ovh
    Description is missing...
    Default value is: false

Extraction
  -Extraction/method -e_m
    Method for extracting unsatisfiable cores with HeadExtraction
    Default value is: VAR
    Possible values: DEC_VAR DEC_CON VAR CON
  -Extraction/nCores -e_n
    Number of unsatifiable cores to be extracted with HeadExtraction
    Default value is: 1
```