Choco3 Documentation

Release 3.3.1

Charles Prud'homme, Jean-Guillaume Fages, Xavier Lorca

November 03, 2015

Contents

Warning: This is a work-in-progress documentation. If you have any questions, suggestions or requests, please send an email to choco3-support@mines-nantes.fr.

Part I

Preliminaries

CHAPTER 1

Main concepts

1.1 What is Constraint Programming?

Such a paradigm takes its features from various domains (Operational Research, Artificial Intelligence, etc). Constraint programming is now part of the portfolio of global solutions for processing real combinatorial problems. Actually, this technique provides tools to deal with a wide range of combinatorial problems. These tools are designed to allow non-specialists to address strategic as well as operational problems, which include problems in planning, scheduling, logistics, financial analysis or bio-informatics. Constraint programming differs from other methods of Operational Research by how it is implemented. Usually, the algorithms must be adapted to the specifications of the problem addressed. This is not the case in Constraint Programming where the problem addressed is described using the tools available in the library. The exercise consists in choosing carefully what constraints combine to properly express the problem, while taking advantage of the benefits they offer in terms of efficiency.

[wikipedia]

1.2 What is Choco ?

Choco is a Free and Open-Source Software dedicated to Constraint Programming. It is written in Java, under BSD license. It aims at describing real combinatorial problems in the form of Constraint Satisfaction Problems and solving them with Constraint Programming techniques.

Choco is used for:

- teaching : easy to use
- research : easy to extend
- real-life applications : easy to integrate

Choco is among the fastest CP solvers on the market. In 2013 and 2014, Choco has been awarded two silver medals and three bronze medals at the MiniZinc challenge that is the world-wide competition of constraint-programming solvers.

In addition to these performance results, Choco benefits from academic contributors, who provide support and long term improvements, and the consulting company COSLING, which provides services ranging from training to the development and the integration of CP models into larger applications.

Choco official website is: http://www.choco-solver.org

1.3 Technical overview

Choco 3 includes:

- various type of variables (integer, boolean, set and real),
- various state-of-the-art constraints (alldifferent, count, nvalues, etc.),
- various search strategies, from basic ones to most complex (impact-based and activity-based search),
- explanation-based engine, that enables conflict-based back jumping, dynamic backtracking and path repair,

But also facilities to interact with the search loop, factories to help modeling, many samples, an interface to Ibex, etc. The source code of choco-solver-3 is hosted on GitHub.

Choco also has many extensions, including a FlatZinc parser to solve minizinc instances and a graph variable module to better solve graph problems such as the TSP.

An overview of the features of Choco 3 may also be found in the presentation made in the "CP Solvers: Modeling, Applications, Integration, and Standardization" workshop of CP2013.

1.4 History

The first version of Choco dates from the early 2000s. A few years later, Choco 2 has encountered a great success in both the academic and the industrial world. For maintenance issue, Choco has been completely rewritten in 2011, leading to Choco 3. The first beta version of Choco 3 has been released in 2012. The latest version is Choco 3.3.1.

1.5 How to get support ?

A forum is available on the website of Choco. It is dedicated to technical questions about the Choco solver and basic modeling helps. If you encounter any bug or would like some features to be added, please feel free to open a discussion on the forum. You can also use the following support mailing list: choco3-support@mines-nantes.fr. As can be seen on the Choco website, most support requests are answered very fast. However, this free service is provided with no guarantee.

If you want an expert to build a CP model for your application or if you need professional support, please contact COSLING.

1.6 How to cite Choco ?

A reference to this manual, or more generally to Choco 3, is made like this:

```
@manual{choco3,
  author        = {Charles Prud'homme and Jean-Guillaume Fages and Xavier Lorca},
  title         = {Choco3 Documentation},
  year          = {2014},
  organization  = {TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.},
  timestamp     = {Thu, 11 May 2015},
  url           = {http://www.choco-solver.org },
}
```

1.7 Who contributes to Choco ?

Core developers	Charles Prud'homme and Jean-Guillaume Fages
Main contributors	Xavier Lorca, Narendra Jussien, Fabien Hermenier, Jimmy Liang.
Previous versions contributors	François Laburthe, Hadrien Cambazard, Guillaume Rochart, Arnaud Malapert, Sophie Demassey, Nicolas Beldiceanu, Julien Menana, Guillaume Richaud, Thierry Petit, Julien Vion, Stéphane Zampelli.

If you want to contribute, let us know.

Choco is developed with Intellij IDEA and JProfiler, that are kindly provided for free.

CHAPTER 2

Getting started

2.1 Installing Choco 3

Choco 3 is a java library based on Java 8. The main library is named `choco-solver` and can be seen as the core library. Some extensions are also provided, such as `choco-parsers` or `choco-cpviz`, and rely on but do not include `choco-solver`.

2.1.1 Which jar to select ?

We provide a zip file which contains the following files:

choco-solver-3.3.1-with-dependencies.jar An ready-to-use jar file including dependencies; it provides tools to declare a Solver, the variables, the constraints, the search strategies, etc. In a few words, it enables modeling and solving CP problems.

choco-solver-3.3.1.jar A jar file excluding all dependencies and configuration file; Enable using choco-solver as a dependency of an application. Otherwise, it provides the same code as the jar with dependencies.

choco-solver-3.3.1-sources.jar The source of the core library.

choco-samples-3.3.1-sources.jar The source of the artifact *choco-samples* made of problems modeled with Choco. It is a good start point to see what it is possible to do with Choco.

apidocs-3.3.1.zip Javadoc of Choco-3.3.1

logback.xml The logback configuration file; may be needed when choco-solver is used as a library.

Please, refer to *README.md* for more details.

Note: Java 7 compliant jars are also available, post-fixed with 'jk7'.

Extensions

There are also official extensions, thus maintained by the Choco team. They are provided apart from the zip file. The available extensions are: *choco-parsers*, *choco-gui*, *choco-cpviz*, *choco-graph*, *choco-geost*, *choco-exppar*. .., *61_ext_eps*.

Note: Each of those extensions include all dependencies but choco-solver classes, which ease their usage.

To start using Choco 3, you need to be make sure that the right version of java is installed. Then you can simply add the `choco-solver` jar file (and extension libraries) to your classpath or declare them as dependency of a Maven-based project.

2.1.2 Update the classpath

Simply add the jar file to the classpath of your project (in a terminal or in your favorite IDE).

2.1.3 As a Maven Dependency

Choco is build and managed using Maven3. Choco is available on Maven Central Repository, to declare Choco as a dependency of your project, simply update the `pom.xml` of your project by adding the following instruction:

```
<dependency>
 <groupId>org.choco-solver</groupId>
 <artifactId>choco-solver</artifactId>
 <version>X.Y.Z</version>
</dependency>
```

where `X.Y.Z` is replaced by 3.3.1. Note that the artifact does not include any dependencies or *logback.xml*. Please, refer to *README.md* for the list of required dependencies.

2.1.4 Compiling sources

As a Maven-based project, Choco can be installed in a few instructions. Once you have downloaded the source (from the zip file or GitHub, simply run the following command:

```
mvn clean install -DskipTests
```

This instruction downloads the dependencies required for Choco3 (such as the trove4j and logback) then compiles the sources. The instruction `-DskipTests` avoids running the tests after compilation (and saves you a couple of hours). Regression tests are run on a private continuous integration server.

Maven provides commands to generate files needed for an IDE project setup. For example, to create the project files for your favorite IDE:

IntelliJ Idea

```
mvn idea:idea
```

Eclipse

```
mvn eclipse:eclipse
```

2.2 Overview of Choco 3

The following steps should be enough to start using Choco 3. The minimal problem should at least contains a solver, some variables and constraints to linked them together.

To facilitate the modeling, Choco 3 provides factories for almost every required component of CSP and its resolution:

Factory	Shortcut	Enables to create
`VariableFactory`	VF	Variables and views (integer, boolean, set and real)
`IntConstraintFactory`	ICF	Constraints over integer variables
`SetConstraintFactory`	SCF	Constraints over set variables
`LogicalConstraintFactory`	LCF	(Manages constraint reification)
`IntStrategyFactory`	ISF	Custom or black-box search strategies
`SetStrategyFactory`	SSF	
`Chatterbox`		Output messages and statistics.
`SearchMonitorFactory`	SMF	resolution limits, restarts etc.

Note that, in order to have a concise and readable model, factories have shortcut names. Furthermore, they can be imported in a static way:

```
import static org.chocosolver.solver.search.strategy.ISF.*;
```

Let say we want to model and solve the following equation: $x + y < 5$, where the $x \in [\![0, 5]\!]$ and $y \in [\![0, 5]\!]$. Here is a short example which illustrates the main steps of a CSP modeling and resolution with Choco 3 to treat this equation.

```
1        // 1. Create a Solver
2        Solver solver = new Solver("my first problem");
3        // 2. Create variables through the variable factory
4        IntVar x = VariableFactory.bounded("X", 0, 5, solver);
5        IntVar y = VariableFactory.bounded("Y", 0, 5, solver);
6        // 3. Create and post constraints by using constraint factories
7        solver.post(IntConstraintFactory.arithm(x, "+", y, "<", 5));
8        // 4. Define the search strategy
9        solver.set(IntStrategyFactory.lexico_LB(x, y));
10       // 5. Launch the resolution process
11       solver.findSolution();
12       //6. Print search statistics
13       Chatterbox.printStatistics(solver);
```

One may notice that there is no distinction between model objects and solver objects. This makes easier for beginners to model and solve problems (reduction of concepts and terms to know) and for developers to implement their own constraints and strategies (short cutting process).

Don't be afraid to take a look at the sources, we think it is a good start point.

2.3 Choco 3 quick documentation

2.3.1 Solver

The `Solver` is a central object and must be created first: `Solver solver = new Solver();`.

[Solver]

2.3.2 Variables

The `VariableFactory` (`VF` for short) eases the creation of variables. Available variables are: `BoolVar`, `IntVar`, `SetVar`, `GraphVar` and `RealVar`. Note, that an `IntVar` domain can be bounded (only bounds are stored) or enumerated (all values are stored); a boolean variable is a 0-1 IntVar.

[Variables]

2.3.3 Views

A view is a variable whose domain is defined by a function over another variable domain. Available views are: `not`, `offset`, `eq`, `minus`, `scale` and `real`.

[Views]

2.3.4 Constants

Fixed-value integer variables should be created with the specific `VF.fixed(int, Solver)` function.

[Constants]

2.3.5 Constraints

Several constraint factories ease the creation of constraints: `LogicalConstraintFactory` (`LCF`), `IntConstraintFactory` (`ICF`) and `SetConstraintsFactory` (`SCF`).

`RealConstraint` is created with a call to new and to `addFunction` method. It requires the Ibex solver.

Constraints hold once posted: `solver.post(c);`

Reified constraints should not be posted.

[Constraints]

2.3.6 Search

Defining a specific way to traverse the search space is made thanks to: `solver.set(AbstractStrategy)`. Predefined strategies are available in `IntStrategyFactory` (`ISF`), `SetStrategyFactory` and `RealStrategyFactory`.

2.3.7 Large Neighborhood Search (LNS)

Various LNS (random, propagation-guided, etc.) can be created from the LNSFactory to improve performance on optimization problems.

2.3.8 Monitors

An `ISearchMonitor` is a callback which enables to react on some resolution events (failure, branching, restarts, solutions, etc.). `SearchMonitorFactory` (`SMF`) lists most useful monitors. User-defined monitors can be added with `solver.plugMonitor(...)`.

2.3.9 Limits

A limit may be imposed on the search. The search stops once a limit is reached. Available limits are `SMF.limitTime(solver, 5000)`, `SMF.limitFail(solver, 100)`, etc.

2.3.10 Restarts

Restart policies may also be applied `SMF.geometrical(...)` and `SMF.luby(...)` are available.

2.3.11 Logging

Logging the search is possible. There are variants but the main way to do it is made through the `Chatterbox.printStatistics(solver)`. It prints the main statistics of the search (time, nodes, fails, etc.)

2.3.12 Solving

Finding if a problem has a solution is made through a call to: `solver.findSolution()`. Looking for the next solution is made thanks to `nextSolution()`. `findAllSolutions()` enables to enumerate all solutions of a problem. To optimize an objective function, call `findOptimalSolution(...)`. Resolutions perform a Depth First Search.

2.3.13 Solutions

By default, the last solution is restored at the end of the search. Solutions can be accessed as they are discovered by using an `IMonitorSolution`.

2.3.14 Explanations

Choco natively supports explained constraints to reduce the search space and to give feedback to the user. Explanations are disabled by default.

2.4 Choco 3 : changes

2.4.1 3.3.2

- **Addition:**
  - *Search loop*
  - *51_icstr_nvpc*
  - *keysorting*
  - *Search Monitors*
  - *dichtotomic*
- **Major modification:**
  - *Multi-thread resolution*

2.4.2 3.3.1

- **Addition:**
  - *Search Monitors*
  - *Defining its own constraint*
- **Major modification:**
  - *Explanations*
  - *Search monitor*

2.4.3 3.3.0

- **Addition:**
  - *Things to know about constraints*
  - *Automaton-based Constraints*
  - *Declaring complex clauses*
  - *Settings*
  - *Search binder*
  - *Resolution statistics*
- **New constraints:**
  - *mddc*
  - *not_member*
- **Major modification:**
  - *Multi-thread resolution*
  - *Defining its own search strategy*

Part II

Modelling problems

CHAPTER 3

The solver

The object `Solver` is the key component. It is built as following:

```
Solver solver = new Solver();
```

or:

```
Solver solver = new Solver("my problem");
```

This should be the first instruction, prior to any other modelling instructions. Indeed, a solver is needed to declare variables, and thus constraints.

Here is a list of the commonly used Solver API.

Note: The API related to resolution are not described here but detailed in *Solving*. Similarly, API provided to add a constraint to the solver are detailed in *Constraints*. The other missing methods are only useful for internal behavior.

3.1 Getters

3.1.1 Variables

Method	Definition
`Variable[] getVars()`	Return the array of variables declared in the solver. It includes all type of variables declared, integer, boolean, etc. but also *fixed* variables such as `Solver.ONE`.
`int getNbVars()`	Return the number of variables involved in the solver.
`Variable getVar(int i)`	Return the i^th variable declared in the solver.
`IntVar[] retrieveIntVars()`	Extract from the solver variables those which are integer (ie whose *KIND* is set to *INT*, that is, including *fixed* integer variables).
`retrieveBoolVars()`	Extract from the solver variables those which are boolean (ie whose *KIND* is set to *BOOL*, that is, including `Solver.ZERO` and `Solver.ONE`).
`SetVar[] retrieveSetVars()`	Extract from the solver variables those which are set (ie whose *KIND* is set to *SET*)
`RealVar[] retrieveRealVars()`	Extract from the solver variables those which are real (ie whose *KIND* is set to *REAL*)
`BoolVar ZERO()`	Return the constant "0".
`BoolVar ONE()`	Return the constant "1".

3.1.2 Constraints

Method	Definition
`Constraint[] getCstrs()`	Return the array of constraints posted in the solver.
`getNbCstrs()`	Return the number of constraints posted in the solver.
`Constraint TRUE()`	Return the basic "true" constraint.
`Constraint FALSE()`	Return the basic "false" constraint.

3.1.3 Other

Method	Definition
`String getName()`	Return the name of the solver.
`IMeasures getMeasures()`	Return a reference to the measure recorder which stores resolution statistics.
`AbstractStrategy getStrategy()`	Return a reference to the declared search strategy.
`Settings getSettings()`	Return the current `Settings` used in the solver.
`ISolutionRecorder getSolutionRecorder()`	Return the solution recorder.
`IEnvironment getEnvironment()`	Return the internal *environment* of the solver, essential to manage backtracking.
`ObjectiveManager getObjectiveManager()`	Return the objective manager of the solver, needed when an objective has to be optimized.
`ExplanationEngine getExplainer()`	Return the explanation engine declared, (default is *NONE*).
`IPropagationEngine getEngine()`	Return the propagation engine of the solver, which *orchestrate* the propagation of constraints.
`ISearchLoop getSearchLoop()`	Return the search loop of the solver, which guide the search process.
`Variable[] getObjectives()`	Return the objective variables.
`double getPrecision()`	In case of real variable to optimize, a precision is required.

3.2 Setters

Method	Definition
`set(Settings settings)`	Set the settings to use while modelling and solving.
`set(AbstractStrategy... strategies)`	Set a strategy to explore the search space. In case many strategies are given, they will be called in sequence.
`set(ISolutionRecorder sr)`	Set a solution recorder, and erase the previous declared one (by default, `LastSolutionRecorder` is declared, it only stores the last solution found.
`set(ISearchLoop searchLoop)`	Set the search loop to use during resolution. The default one is a binary search loop.
`set(IPropagationEngine propagationEngine)`	Set the propagation engine to use during resolution. The default one is `SevenQueuesPropagatorEngine`.
`set(ExplanationEngine explainer)`	Set the explanation engine to use during resolution. The default one is `ExplanationEngine` which does nothing.
`set(ObjectiveManager om)`	Set the objective manager to use during the resolution. *For advanced usage only*.
`setObjectives(Variable... objectives)`	Define the variables to optimize (most of the time, only one variable is declared).
`set(ObjectiveManager om)`	In case of real variable to optimize, a precision is required.

3.3 Others

Method	Definition
`Solver duplicateModel()`	Duplicate the model associated with a solver, ie only variables and constraints, and return a new solver.
`void makeCompleteSearch(boolean isComplete)`	Add a strategy to the declared one in order to ensure that all variables are covered by (at least) one strategy.
`void plugMonitor(ISearchMonitor sm)`	Put a *search monitor* to react on search events (solutions, decisions, fails, ...).
`void plugMonitor(FilteringMonitor fm)`	Add an filtering monitor, that is an object that is kept informed of all (propagation) events generated during the resolution.

CHAPTER 4

Declaring variables

4.1 Principle

A variable is an *unknown*, mathematically speaking. The goal of a resolution is to *assign* a *value* to each declared variable. In Constraint Programming, the *domain* –set of values that a variable can initially take– must be defined.

Choco 3 includes five types of variables: `IntVar`, `BoolVar`, `SetVar` and `RealVar`. A factory is available to ease the declaration of variables: `VariableFactory` (or `VF` for short). At least, a variable requires a name and a solver to be declared in. The name is only helpful for the user, to read the results computed.

4.2 Integer variable

An integer variable is based on domain made with integer values. There exists under three different forms: **bounded**, **enumerated** or **boolean**. An alternative is to declare variable-based views.

Important: It is strongly recommended to not define unbounded domain like `[Integer.MIN_VALUE, Integer.MAX_VALUE]`. Indeed, such domain definition may lead to :

- incorrect domain size (`Integer.MAX_VALUE - Integer.MIN_VALUE +1 = 0`)
- and to numeric overflow/underflow operations during propagation.

If *undefined* domain is really required, the following range should be considered: `[VariableFactory.MIN_INT_BOUND, VariableFactory.MAX_INT_BOUND]`. Such an interval defines *42949673* values, from *-21474836* to *21474836*.

4.2.1 Bounded variable

Bounded (integer) variables take their value in $[\![a, b]\!]$ where a and b are integers such that $a < b$ (the case where $a = b$ is handled through views). Those variables are pretty light in memory (the domain requires two integers) but cannot represent holes in the domain.

To create a bounded variable, the `VariableFactory` should be used:

```
IntVar v = VariableFactory.bounded("v", 1, 12, solver);
```

To create an array of 5 bounded variables of initial domain $[\![-2, 8]\!]$:

```
IntVar[] vs = VariableFactory.boundedArray("vs", 5, -2, 8, solver);
```

To create a matrix of 5x6 bounded variables of initial domain $[\![0, 5]\!]$:

```
IntVar[][] vs = VariableFactory.boundedMatrix("vs", 5, 6, 0, 5, solver);
```

Note: When using bounded variables, branching decisions must either be domain splits or bound assignments/removals. Indeed, assigning a bounded variable to a value strictly comprised between its bounds may results in disastrous performances, because such branching decisions will not be refutable.

4.2.2 Enumerated variable

Integer variables with enumerated domains, or shortly, enumerated variables, take their value in $[\![a, b]\!]$ where a and b are integers such that $a < b$ (the case where $a = b$ is handled through views) or in an array of ordered values $a, b, c, .., z$, where $a < b < c... < z$. Enumerated variables provide more information than bounded variables but are heavier in memory (usually the domain requires a bitset).

To create an enumerated variable, the `VariableFactory` should be used:

```
IntVar v = VariableFactory.enumerated("v", 1, 12, solver);
```

which is equivalent to :

```
IntVar v = VariableFactory.enumerated("v", new int[]{1,2,3,4,5,6,7,8,9,10,11,12}, solver);
```

To create a variable with holes in its initial domain:

```
IntVar v = VariableFactory.enumerated("v", new int[]{1,7,8}, solver);
```

To create an array of 5 enumerated variables with same domains:

```
IntVar[] vs = VariableFactory.enumeratedArray("vs", 5, -2, 8, solver);

IntVar[] vs = VariableFactory.enumeratedArray("vs", 5, new int[]{-10, 0, 10}, solver);
```

To create a matrix of 5x6 enumerated variables with same domains:

```
IntVar[][] vs = VariableFactory.enumeratedMatrix("vs", 5, 6, 0, 5, solver);

IntVar[][] vs = VariableFactory.enumeratedMatrix("vs", 5, 6, new int[]{1,2,3,5,6,99}, solver);
```

Modelling: Bounded or Enumerated?

The choice of representation of the domain variables should not be done lightly. Not only the memory consumption should be considered but also the used constraints. Indeed, some constraints only update bounds of integer variables, using them with bounded variables is enough. Others make holes in variables' domain, using them with enumerated variables takes advantage of the *power* of the filtering algorithm. Most of the time, variables are associated with propagators of various *power*. The choice of domain representation must then be done on a case by case basis.

4.2.3 Boolean variable

Boolean variables, BoolVar, are specific `IntVar` which take their value in $[\![0, 1]\!]$.

To create a new boolean variable:

```
BoolVar b = VariableFactory.bool("b", solver);
```

To create an array of 5 boolean variables:

```
BoolVar[] bs = VariableFactory.boolArray("bs", 5, solver);
```

To create a matrix of 5x6 boolean variables:

```
BoolVar[] bs = VariableFactory.boolMatrix("bs", 5, 6, solver);
```

4.3 Constants

Fixed-value integer variables should be created with a call to the following functions:

```
VariableFactory.fixed("seven", 7, solver);
```

Or:

```
VariableFactory.fixed(8, Solver)
```

where 7 and 8 are the constant values. Not specifying a name to a constant enables the solver to use *cache* and avoid multiple occurrence of the same object in memory.

4.4 Variable views

Views are particular integer variables, they can be used inside constraints. Their domains are implicitly defined by a function and implied variables.

`x` is a constant :

```
IntVar x = VariableFactory.fixed(1, solver);
```

`x = y + 2` :

```
IntVar x = VariableFactory.offset(y, 2);
```

`x = -y` :

```
IntVar x = VariableFactory.minus(y);
```

`x = 3*y` :

```
IntVar x = VariableFactory.scale(y, 3);
```

Views can be combined together:

```
IntVar x = VariableFactory.offset(VariableFactory.scale(y,2),5);
```

4.5 Set variable

A set variable `SV` represents a set of integers. Its domain is defined by a set interval: `[S_E,S_K]`

- the envelope `S_E` is an `ISet` object which contains integers that potentially figure in at least one solution,
- the kernel `S_K` is an `ISet` object which contains integers that figure in every solution.

Initial values for both `S_K` and `S_E` can be specified. If no initial value is given for `S_K`, it is empty by default. Then, decisions and filtering algorithms will remove integers from `S_E` and add some others to `S_K`. A set variable is instantiated if and only if `S_E = S_K`.

A set variable can be created as follows:

```
// z initial domain
int[] z_envelope = new int[]{2,1,3,5,7,12};
int[] z_kernel = new int[]{2};
z = VariableFactory.set("z", z_envelope, z_kernel, solver);
```

4.6 Real variable

Real variables have a specific status in Choco 3. Indeed, continuous variables and constraints are managed with Ibex solver.

A real variable is declared with two doubles which defined its bound:

```
RealVar x = VariableFactory.real("y", 0.2d, 1.0e8d, 0.001d, solver);
```

Or a real variable can be declared on the basis of on integer variable:

```
IntVar ivar = VariableFactory.bounded("i", 0, 4, solver);
RealVar x = VariableFactory.real(ivar, 0.01d);
```

CHAPTER 5

Constraints and propagators

5.1 Principle

A constraint is a logic formula that defines allowed combinations of values for its variables, that is, restrictions over variables that must be respected in order to get a feasible solution. A constraint is equipped with a (set of) filtering algorithm(s), named *propagator(s)*. A propagator **removes**, from the domains of the targeted variables, values that cannot correspond to a valid combination of values. A solution of a problem is an assignment of all its variables simultaneously verifying all the constraints.

Constraint can be declared in *extension*, by defining the valid/invalid tuples, or in *intension*, by defining a relation between the variables. Choco 3 provides various factories to declare constraints (see *Overview* to have a list of available factories). A list of constraints available through factories is given in *List of available constraints*.

Modelling: Selecting the right constraints

Constraints, through propagators, suppress forbidden values of the domain of the variables. For a given paradigm, there can be several propagators available. A widely used example is the *AllDifferent* constraints which holds that all its variables should take a distinct value in a solution. Such a rule can be formulated :

- using a clique of inequality constraints,
- using a global constraint: either analysing bounds of variable (*Bound consistency*) or analysing all values of the variables (*Arc consistency*),
- or using a table constraint –an extension constraint which list the valid tuples.

The choice must be made by not only considering the gain in expressiveness of stress compared to others. Indeed, the effective yield of each option can be radically different as the efficiency in terms of computation time.

Many global constraints are used to model problems that are inherently NP-complete. And only a partial domain filtering variables can be done through a polynomial algorithm. This is for example the case of *NValue* constraint that one aspect relates to the problem of "minimum hitting set." Finally, the *global* nature of this type of constraint also simplifies the work of the solver in that it provides all or part of the structure of the problem.

If we want an integer variable `sum` to be equal to the sum of values of variables in the set `atLeast`, we can use the `IntConstraintFactory.sum` constraint:

```
            }
```

A constraint may define its specific checker through the method `isSatisfied()`, but most of the time the checker is given by checking the entailment of each of its propagators. The satisfaction of the constraints' solver is done on each solution if assertions are enabled.

Note: One can enable assertions by adding the `-ea` instruction in the JVM arguments.

It can thus be slower if the checker is often called (which is not the case in general). The advantage of this framework is the economy of code (less constraints need to be implemented), the avoidance of useless redundancy when several constraints use the same propagators (for instance `IntegerChanneling` constraint involves `AllDifferent constraint`), which leads to better performances and an easier maintenance.

Note: To ease modelling, it is not required to manipulate propagators, but only constraints. However, one can define specific constraints by defining combinations of propagators and/or its own propagators. More detailed are given in *Defining its own constraint*.

Choco 3 provides various types of constraints.

5.1.1 Available constraints

FALSE, *TRUE*

On one integer variable

arithm, *member*, *not_member*.

On two integer variables

absolute, *arithm*, *distance*, *square*, *table*, *times*.

On three integer variables

arithm, *distance*, *eucl_div*, *maximum*, *minimum*, *mod*, *times*.

On an undefined number of integer variables

element, *sort*, *keysorting*, *table*, *mddc*.

alldifferent, *alldifferent_conditionnal*, *alldifferent_except_0*, *global_cardinality*,

among, *atleast_nvalues*, *atmost_nvalues*, *count*, *nvalues*,

boolean_channeling, *clause_channeling*, *inverse_channeling*.

cumulative, *diffn*.

lex_chain_less, *lex_chain_less_eq*, *lex_less*, *lex_less_eq*, *51_icstr_nvpc*.

maximum, *minimum*,

scalar, *sum*.

cost_regular, *multicost_regular*, *regular*.

circuit, *path*, *subcircuit*, *subpath*, *tree*.

bin_packing, *knapsack*, *tsp*.

On one set variable

notEmpty.

On two set variables

disjoint, *offSet*.

On an undefined number of set variables

all_different, *all_disjoint*, *all_equal*, *bool_channel*, *intersection*, *inverse_set*, *member*, *nbEmpty*, *partition*, *subsetEq*, *symmetric*, *union*.

On integer and set variables

cardinality, *element*, *int_channel*, *max*, *member*, *not_member*, *min*, *sum*.

On real variables

Constraints over real variables.

5.1.2 Things to know about constraints

Automaton-based Constraints

cost_regular, *multicost_regular* and *regular* rely on an automaton, declared implicitly or explicitly. There are two kinds of `IAutomaton` : `FiniteAutomaton`, needed for *cost_regular*, and *CostAutomaton*, required for *multicost_regular* and *regular*. A `CostAutomaton` is an extension of `FiniteAutomaton` where costs can be declared per transition.

`FiniteAutomaton` embeds an `Automaton` object provided by the `dk.brics.automaton` library. Such an automaton accepts fixed-size words made of multiple `char` s, but the regular constraints rely on `IntVar` s. So, mapping between `char` (needed by the underlying library) and `int` (declared in `IntVar`) is made. The mapping enables declaring regular expressions where a symbol is not only a digit between *0* and *9* but any positive number. Then to distinct, in the word *101*, the symbols *0*, *1*, *10* and *101*, two additional `char` are allowed in a regexp: < and > which delimits numbers.

In summary, a valid regexp for the *cost_regular*, *multicost_regular* and *regular* constraints is a combination of **digits** and Java Regexp special characters.

Examples of allowed RegExp

`"0*11111110+10+10+11111110*"`, `"11(0|1|2)*00"`, `"(0|<10> |<20>)*(0|<10>)"`.

Example of forbidden RegExp

`"abc(a|b|c)*"`.

5.2 Posting constraints

To be effective, a constraint must be posted to the solver. This is achieved using the method:

```
solver.post(Constraint cstr);
```

Otherwise, if the `solver.post(Constraint cstr)` method is not called, the constraint will not be taken into account during the resolution process : it may not be satisfied in all solutions.

Method	Definition
`void post(Constraint c)`	Post **permanently** a constraint in the constraint network defined by the solver. The constraint is not propagated on posting, but is added to the propagation engine.
`void post(Constraint... cs)`	Post **permanently** the constraints in the constraint network defined by the solver.
`void postTemp(Constraint c)`	Post a constraint **temporary** in the constraint network. The constraint will active on the current sub-tree and be removed upon backtrack.
`void unpost(Constraint c)`	Remove permanently the constraint from the constraint network

5.3 Reifying constraints

In Choco 3, it is possible to reify any constraint. Reifying a constraint means associating it with a `BoolVar` to represent whether the constraint holds or not:

```
BoolVar b = constraint.reify();
```

Or:

```
BoolVar b = VF.bool("b", solver);
constraint.reifyWith(b);
```

The first API `constraint.reify()` creates the variable, if it does not already exists, and reify the constraint. The second API `constraint.reifyWith(b)` reify the constraint with the given variable.

Note: A constraint is reified with only one boolean variable. If multiple reification are required, equality constraints will be created.

The `LogicalConstraintFactory` enables to manipulate constraints through their reification.

Reifying a constraint means that we allow the constraint not to be satisfied. Therefore, the reified constraint **should not** be posted. Only the reifying constraint should be posted. Note also that, for performance reasons, some reifying constraints available in the `LogicalConstraintFactory` are **automatically posted** (the factory method returns void).

For instance, we can represent the constraint "either `x<0` or `y>42`" as the following:

```
Constraint a = IntConstraintFactory.arithm(x,"<",0);
Constraint b = IntConstraintFactory.arithm(y,">",42);
Constraint c = LogicalConstraintFactory.or(a,b);
solver.post(c);
```

This will actually reify both constraints `a` and `b` and say that at least one of the corresponding boolean variables must be true. Note that only the constraint `c` is posted.

As a second reification example, let us consider "if `x<0` then `y>42`":

```
Constraint a = IntConstraintFactory.arithm(x,"<",0);
Constraint b = IntConstraintFactory.arithm(y,">",42);
LogicalConstraintFactory.ifThen(a,b);
```

This time the `LogicalConstraintFactory.ifThen` returns void, meaning that the constraint is automatically posted. If one really needs to access an `ifThen Constraint` object, then the `LogicalConstraintFactory.ifThen_reifiable` method should be used instead.

5.4 SAT constraints

A SAT solver is embedded in Choco. It is not designed to be accessed directly. The SAT solver is internally managed as a constraint (and a propagator), that's why it is referred as SAT constraint in the following.

Important: The SAT solver is directly inspired by MiniSat[EenSorensson03]. However, it only propagates clauses, no learning or search is implemented.

On a call to any methods of `solver.constraints.SatFactory`, the SAT constraint (and propagator) is created and automatically posted to the solver.

5.4.1 How to add clauses

Clauses can be added with calls to the `SatFactory`.

On one boolean variable

addTrue, *addFalse*.

On two boolean variables

addBoolEq, *addBoolLe*, *addBoolLt*, *addBoolNot*.

Reification on two boolean variables

addBoolIsEqVar, *addBoolIsLeVar*, *addBoolIsLtVar*, *addBoolIsNeqVar*, *addBoolOrEqVar*, *addBoolXorEqVar*, *addBoolAndEqVar*.

On undefined number of boolean variables

addBoolOrArrayEqualTrue, *addAtMostNMinusOne*, *addAtMostOne*, *addBoolAndArrayEqualFalse*, *addBoolAndArrayEqVar*, *addBoolOrArrayEqVar*, *addClauses*, *addMaxBoolArrayLessEqVar*, *addSumBoolArrayGreaterEqVar*, *addSumBoolArrayLessEqVar*.

Even if `SatFactory` should be the first factory to use when dealing with clauses (see for instance *addClauses* to create complex expressions), there exists a alternative: *Logical constraints*.

5.4.2 Declaring complex clauses

There is a convenient way to declare complex clauses by calling *addClauses*. The method takes a `LogOp` and an instance of `Solver` as input, extracts the underlying clauses and add them to the `SatFactory`.

A `LogOp` is an implementation of `ILogical`, just like `BoolVar`, and provides the following API:

`LogOp and(ILogical... operands)` : create a conjunction, results in *true* if all of its operands are *true*.

`LogOp ifOnlyIf(ILogical a, ILogical b)`: create a biconditional, results in *true* if and only if both operands are false or both operands are *true*.

`LogOp ifThenElse(ILogical a, ILogical b, ILogical c)` : create an implication, results in *true* if `a` is *true* and `b` is *true* or `a` is `false` and ``c` is *true*.

`LogOp implies(ILogical a, ILogical b)` : create an implication, results in *true* if `a` is *false* or `b` is *true*.

`LogOp reified(BoolVar b, ILogical tree)` : create a logical connection between `b` and `tree`.

`LogOp or(ILogical... operands)` : create a disjunction, results in *true* whenever one or more of its operands are *true*.

`LogOp nand(ILogical... operands)` : create an alternative denial, results in if at least one of its operands is *false*.

`LogOp nor(ILogical... operands)` : create a joint denial, results in *true* if all of its operands are *false*.

`LogOp xor(ILogical a, ILogical b)` : create an exclusive disjunction, results in *true* whenever both operands differ.

`ILogical negate(ILogical l)` : return the logical complement of *l*.

The resulting logical operation can be very verbose, but certainly more easy to declare:

```
SatFactory.addClauses(LogOp.and(LogOp.nand(LogOp.nor(a, b), LogOp.or(c, d)), e));
SatFactory.addClauses(LogOp.nor(LogOp.or(LogOp.nand(a, b), c), d));
SatFactory.addClauses(LogOp.and(LogOp.nand(LogOp.nor(a, b), LogOp.or(c, d)), e));
```

Part III

Solving problems

CHAPTER 6

Finding solutions

Choco 3 provides different API, offered by `Solver`, to launch the problem resolution. Before everything, there are two methods which help interpreting the results.

Feasibility: Once the resolution ends, a call to the `solver.isFeasible()` method will return a boolean which indicates whether or not the problem is feasible.

- `true`: at least one solution has been found, the problem is proven to be feasible,
- `false`: in principle, the problem has no solution. More precisely, if the search space is guaranteed to be explored entirely, it is proven that the problem has no solution.

Limitation: When the resolution is limited (See *Limiting the resolution* for details and examples), one may guess if a limit has been reached. The `solver.hasReachedLimit()` method returns `true` if a limit has bypassed the search process, `false` if it has ended *naturally*.

Warning: In some cases, the search may not be complete. For instance, if one enables restart on each failure with a static search strategy, there is a possibility that the same sub-tree is explored permanently. In those cases, the search may never stop or the two above methods may not be sufficient to confirm the lack of solution.

6.1 Satisfaction problems

6.1.1 Finding a solution

A call to `solver.findSolution()` launches a resolution which stops on the first solution found, if any.

```
1  public void overview1() {
2      // 1. Create a Solver
3      Solver solver = new Solver("my first problem");
4      // 2. Create variables through the variable factory
5      IntVar x = VariableFactory.bounded("X", 0, 5, solver);
6      IntVar y = VariableFactory.bounded("Y", 0, 5, solver);
7      // 3. Create and post constraints by using constraint factories
8      solver.post(IntConstraintFactory.arithm(x, "+", y, "<", 5));
9      // 4. Define the search strategy
10     solver.set(IntStrategyFactory.lexico_LB(x, y));
11     // 5. Launch the resolution process
12     solver.findSolution();
13     //6. Print search statistics
```

If a solution has been found, the resolution process stops on that solution, thus each variable is instantiated to a value, and the method returns `true`.

If the method returns `false`, two cases must be considered:

- A limit has been reached. There may be a solution, but the solver has not been able to find it in the given limit or there is no solution but the solver has not been able to prove it (i.e., to close to search tree) in the given limit. The resolution process stops in no particular place in the search tree and the resolution can be run again.
- No limit has been declared. The problem has no solution, the complete exploration of the search tree proved it.

To ensure the problem has no solution, one may call `solver.hasReachedLimit()`. It returns `true` if a limit has been reached, `false` otherwise.

6.1.2 Enumerating solutions

Once the resolution has been started by a call to `solver.findSolution()` and if the problem is feasible, the resolution can be resumed using `solver.nextSolution()` from the last solution found. The method returns `true` if a new solution is found, `false` otherwise (a call to `solver.hasReachedLimit()` must confirm the lack of new solution). If a solution has been found, alike `solver.findSolution()`, the resolution stops on this solution, each variable is instantiated, and the resolution can be resumed again until there is no more new solution.

One may enumerate all solution like this:

```
if(solver.findSolution()){
   do{
       // do something, e.g. print out variables' value
   }while(solver.nextSolution());
}
```

`solver.findSolution()` and `solver.nextSolution()` are the only ways to resume a resolution process which has already began.

Tip: On a solution, one can get the value assigned to each variable by calling

```
ivar.getValue(); // instantiation value of an IntVar, return a int
svar.getValues(); // instantiation values of a SerVar, return a int[]
rvar.getLB(); // lower bound of a RealVar, return a double
rvar.getUB(); // upper bound of a RealVar, return a double
```

An alternative is to call `solver.findAllSolutions()`. It attempts to find all solutions of the problem. It returns the number of solutions found (in the given limit if any).

6.2 Optimization problems

Choco 3 enables to solve optimization problems, that is, in which a variable must be optimized.

Tip: For functions, one should declare an objective variable and declare it as the result of the function:

```
// Function to maximize: 3X + 4Y
IntVar OBJ = VF.bounded("objective", 0, 999, solver);
solver.post(ICF.scalar(new IntVar[]{X,Y}, new int[]{3,4}, OBJ));
solver.findOptimalSolution(ResolutionPolicy.MAXIMIZE, OBJ);
```

6.2.1 Finding one optimal solution

Finding one optimal solution is made through a call to the `solver.findOptimalSolution(ResolutionPolicy, IntVar)` method. The first argument defines the kind of optimization required: minimization (`ResolutionPolicy.MINIMIZE`) or maximization (`ResolutionPolicy.MAXIMIZE`). The second argument indicates the variable to optimize.

For instance:

```
solver.findOptimalSolution(ResolutionPolicy.MAXIMIZE, OBJ);
```

states that the variable `OBJ` must be maximized.

The method does not return any value. However, the best solution found so far is restored.

Important: Because the best solution is restored, all variables are instantiated after a call to `solver.findOptimalSolution(...)`.

The best solution found is the optimal one if the entire search space has been explored.

The process is the following: anytime a solution is found, the value of the objective variable is stored and a *cut* is posted. The cut is an additional constraint which states that the next solution must be strictly better than the current one, ie in minimization, strictly smaller.

6.2.2 Finding all optimal solutions

There could be more than one optimal solutions. To find them all, one can call `findAllOptimalSolutions(ResolutionPolicy, IntVar, boolean)`. The two first arguments defines the optimisation policy and the variable to optimize. The last argument states the way the solutions are computed. Set to `true` the resolution will be achieved in two steps: first finding and proving an optimal solution, then enumerating all solutions of optimal cost. Set to `false`, the posted cuts are *soft*. When an equivalent solution is found, it is stored and the resolution goes on. When a strictly better solution is found, previous solutions are removed. Setting the boolean to `false` allow finding non-optimal intermediary solutions, which may be time consuming.

6.3 Multi-objective optimization problems

6.3.1 Finding the pareto front

It is possible to solve a multi-objective optimization problems with Choco 3, using `solver.findParetoFront(ResolutionPolicy policy, IntVar... objectives)`.
The first argument define the resolution policy, which can be `Resolution.MINIMIZE` or `ResolutionPolicy.MAXIMIZE`. Then, the second argument defines the list of variables to optimize.

Note: All variables should respect the same resolution policy.

The underlying approach is naive, but it simplifies the process. Anytime a solution is found, a cut is posted which states that at least one of the objective variables must be better. Such as $(X_0 < b_0 \vee X_1 < b_1 \vee \ldots \vee X_n < b_n$ where X_i is the ith objective variable and b_i its best known value.

The method ends by restoring the last solution found so far, if any.

Here is a simple illustration:

6.4 Propagation

One may want to propagate each constraint manually. This can be achieved by calling `solver.propagate()`. This method runs, in turn, the domain reduction algorithms of the constraints until it reaches a fix point. It may throw a `ContradictionException` if a contradiction occurs. In that case, the propagation engine must be flushed calling `solver.getEngine().flush()` to ensure there is no pending events.

> **Warning:** If there are still pending events in the propagation engine, the propagation may results in unexpected results.

CHAPTER 7

Recording solutions

Choco 3 requires that each decision variable (that is, which is declared in the search strategy) is instantiated in a solution. Otherwise, an exception will be thrown. Non decision variables can be uninstantiated in a solution, however, if WARN logging is enable, a trace is shown to inform the user. Choco 3 includes several ways to record solutions, the recommended way is to plug a *ISolutionMonitor* in. See *Search monitor* for more details.

7.1 Solution storage

A solution is usually stored through a `Solution` object which maps every variable with its current value. Such an object can be erased to store new solutions.

7.2 Solution recording

7.2.1 Built-in solution recorders

A solution recorder (`ISolutionRecorder`) is an object in charge of recording variable values in solutions. There exists many built-in solution recorders:

`LastSolutionRecorder` only keeps variable values of the last solution found. It is the default solution recorder. Furthermore, it is possible to restore that solution after the search process ends. This is used by default when seeking an optimal solution.

`AllSolutionsRecorder` records all solutions that are found. As this may result in a memory explosion, it is not used by default.

`BestSolutionsRecorder` records all solutions but removes from the solution set each solution that is worse than the best solution value found so far. This may be used to enumerate all optimal (or at least, best) solutions of a problem.

`ParetoSolutionsRecorder` records all solutions of the pareto front of the multi-objective problem.

7.2.2 Custom recorder

You can build you own way of manipulating and recording solutions by either implementing your own `ISolutionRecorder` object or by simply using an `ISolutionMonitor`, as follows:

7.3 Solution restoration

A `Solution` object can be restored, i.e. variables are fixed back to their values in that solution. For this purpose, we recommend to restore initial domains and then restore the solution, with the following code:

```
try{
   solver.getSearchLoop().restoreRootNode();
   solver.getEnvironment().worldPush();
   solution.restore();
}catch (ContradictionException e){
   throw new UnsupportedOperationException("restoring the solution ended in a failure");
}
solver.getEngine().flush();
```

Note that if initial domains are not restored, then the solution restoration may lead to a failure. This would happen when trying to restore out of the current domain.

CHAPTER 8

Search Strategies

8.1 Principle

The search space induces by variables' domain is equal to $S = |d_1| * |d_2| * ... * |d_n|$ where d_i is the domain of the i^{th} variable. Most of the time (not to say always), constraint propagation is not sufficient to build a solution, that is, to remove all values but one from (integer) variables' domain. Thus, the search space needs to be explored using one or more *search strategies*. A search strategy performs a performs a Depth First Search and reduces the search space by making *decisions*. A decision involves a variables, a value and an operator, for instance $x = 5$. Decisions are computed and applied until all the variables are instantiated, that is, a solution is found, or a failure has been detected.

> Choco 3.3.1 build a binary search tree: each decision can be refuted. When a decision has to be computed, the search strategy is called to provide one, for instance $x = 5$. The decision is then applied, the variable, the domain of `x` is reduced to `5`, and the decision is validated thanks to the propagation. If the application of the decision leads to a failure, the search backtracks and the decision is refuted ($x \neq 5$) and validated through propagation. Otherwise, if there is no more free variables then a solution has been found, else a new decision is computed.

Note: There are many ways to explore the search space and this steps should not be overlooked. Search strategies or heuristics have a strong impact on resolution performances. Thus, it is strongly recommended to adapt the search space exploration to the problem treated.

8.2 Zoom on IntStrategy

A search strategy `IntStrategy` is dedicated to `IntVar` only. It is based on a list of variables `scope`, a selector of variable `varSelector`, a value selector `valSelector` and an optional `decOperator`.

1. `scope`: array of variables to branch on.
2. `varSelector`: a variable selector, defines how to select the next variable to branch on.
3. `valSelector`: a value selector, defines how to select a value in the domain of the selected variable.
4. `decOperator`: a decision operator, defines how to modify the domain of the selected variable with the selected value.

On a call to `IntStrategy.getDecision()`, `varSelector` try to find, among `scope`, a variable not yet instantiated. If such a variable does not exist, the method returns `null`, saying that it can not compute decision anymore. Otherwise, `valSelector` selects a value, within the domain of the selected variable. A decision can then be computed with the selected variable and the selected value, and is returned to the caller.

By default, the decision built is an assignment: its application leads to an instantiation, its refutation, to a value removal. It is possible create other types of decision by defining a decision operator `DecisionOperator`.

API

IntStrategyFactory.custom(VariableSelector<IntVar> VAR_SELECTOR, IntValueSelector VAL_SELECTOR, DecisionOperator<IntVar> DEC_OPERATOR, IntVar... VARS)

IntStrategyFactory.custom(VariableSelector<IntVar> VAR_SELECTOR, IntValueSelector VAL_SELECTOR, IntVar... VARS)

new IntStrategy(IntVar[] scope, VariableSelector<IntVar> varSelector, IntValueSelector valSelector) new IntStrategy(IntVar[] scope, VariableSelector<IntVar> varSelector, IntValueSelector valSelector,

DecisionOperator<IntVar> decOperator)

Sometimes, on a call to the variable selector, several variables could be selected. In that case, the order induced by `VARS` is used to break tie: the variable with the smallest index is selected. However, it is possible to break tie with other `VAR_SELECTOR``s. They should be declared as parameters of ``VariablesSelectorWithTies`.

```
solver.set(ISF.custom(
    new VariableSelectorWithTies(new FirstFail(), new Random(123L)),
    new IntDomainMin(), vars);
```

The variable with the smallest domain is selected first. If there are more than one variable whose domain size is the smallest, ties are randomly broken.

Note: Only variable selectors which implement `VariableEvaluator` can be used to break ties.

Very similar operations are achieved in `SetStrategy` and `RealStrategy`.

See `solver.search.strategy.IntStrategyFactory` and `solver.search.strategy.SetStrategyFactory` for built-in strategies and selectors.

8.2.1 Available variable selectors

For integer variables

lexico_var_selector, *random_var_selector*, *minDomainSize_var_selector*, *maxDomainSize_var_selector*, *maxRegret_var_selector*.

For set variables

See `solver.search.strategy.selectors.variables.MaxDelta`, `solver.search.strategy.selectors.va`

For real variables

See `solver.search.strategy.selectors.variables.Cyclic`.

8.2.2 Available value selectors

For integer variables

min_value_selector, *mid_value_selector*, *max_value_selector*, *randomBound_value_selector*, *random_value_selector*.

For set variables

See `solver.search.strategy.selectors.values.SetDomainMin`.

For real variables

See `solver.search.strategy.selectors.values.RealDomainMiddle`,
`solver.search.strategy.selectors.values.RealDomainMin`
`solver.search.strategy.selectors.values.RealDomainMax`.

8.2.3 Available decision operators

assign, *remove*, *split*, *reverse_split*.

8.2.4 Available strategies

For integer variables

custom, *dichtotomic*, *once*, *sequencer*.

lexico_LB, *lexico_Neq_LB*, *lexico_Split*, *lexico_UB*, *minDom_LB*, *minDom_MidValue*, *maxDom_Split*, *minDom_UB*, *maxReg_LB*, *objective_bottom_up*, *objective_dichotomic*, *objective_top_bottom*, *random_bound*, *random_value*.

domOverWDeg, *activity*, *impact*.

lastConflict.

51_sstrat_gat.

For set variables

custom, *sequencer*.

force_first, *force_maxDelta_first*, *force_minDelta_first*, *remove_first*.

lastConflict.

Important: Black-box search strategies

There are many ways of choosing a variable and computing a decision on it. Designing specific search strategies can be a very tough task to do. The concept of *Black-box search heuristic* (or adaptive search strategy) has naturally emerged from this statement. Most common black-box search strategies observe aspects of the CSP resolution in order to drive the variable selection, and eventually the decision computation (presumably, a value assignment). Three main families of heuristic, stemming from the concepts of variable impact, conflict and variable activity, can be found in Chocolrelease|. Black-box strategies can be augmented with restarts.

8.3 Default search strategies

If no search strategy is specified in the model, Choco 3 will rely on the default one (defined by a `DefaultSearchBinder` in `Settings`). In many cases, this strategy will not be sufficient to produce satisfying performances and it will be necessary to specify a dedicated strategy, using `solver.set(...)`. The default search strategy distinguishes variables per types and defines a specific search strategy per each type, sequentially applied:

1. integer variables and boolean variables : `IntStrategyFactory.domOverWDeg(ivars, 0)`
2. set variables: `SetStrategyFactory.force_minDelta_first(svars)`
3. real variables `RealStrategyFactory.cyclic_middle(rvars)`

4. objective variable, if any: lower bound or upper bound, depending on the *ResolutionPolicy*

Note that *ISF.lastConflict(solver)* is also plugged-in. Constants are excluded from search strategies' variable scope and the creation order is maintained per types.

`IntStrategyFactory`, `SetStrategyFactory` and `RealStrategyFactory` offer several built-in search strategies and a simple framework to build custom searches.

8.3.1 Search binder

It is possible to override the default search strategy by implementing an `ISearchBinder`. By default, a `Solver` is created with a `DefaultSearchBinder` declared in its settings.

An `ISearchBinder` has the following API:

`void configureSearch(Solver solver)` Configure the search strategy, and even more, of the given solver. The method is called from the search loop, after the initial propagation, if no search strategy is defined. Otherwise, it should be called before running the resolution.

The search binder to use must be declared in the `Setting` attached to a `Solver` (see *Settings*).

8.4 Composition of strategies

Most of the time, it is necessary to combine various strategies. A `StrategiesSequencer` enables to compose various `AbstractStrategy`. It is created on the basis of a list of `AbstractStrategy`. The current active strategy is called to compute a decision through its `getDecision()` method. When no more decision can be computed for the current strategy, the following one becomes active. The intersection of variables from each strategy does not have to be empty. When a variable appears in various strategy, it is ignored as soon as it is instantiated.

When no environment is given in parameter, the last active strategy is not stored, and strategies are evaluated in lexicographical order to find the first active one, based on its capacity to return a decision.

When an environment is given in parameter, the last active strategy is stored.

API

IntStrategyFactory.sequencer(AbstractStrategy... strategies)

Note that a strategy sequencer is automatically generated when setting multiple strategies at the same time:

`solver.set(strategy1,strategy2);` is equivalent to

```
solver.set(ISF.sequencer(strategy1,strategy2));
```

Finally, one can create its own strategy, see *Defining its own search* for more details.

8.5 Restarts

Restart means stopping the current tree search, then starting a new tree search from the root node. Restarting makes sense only when coupled with randomized dynamic branching strategies ensuring that the same enumeration tree is not constructed twice. The branching strategies based on the past experience of the search, such as adaptive search strategies, are more accurate in combination with a restart approach.

Unless the number of allowed restarts is limited, a tree search with restarts is not complete anymore. It is a good strategy, though, when optimizing an NP-hard problem in a limited time.

Some adaptive search strategies resolutions are improved by sometimes restarting the search exploration from the root node. Thus, the statistics computed on the bottom of the tree search can be applied on the top of it.

There a two restart strategies available in `SearchMonitorFactory`:

```
geometrical(Solver solver, int base, double grow, ICounter counter, int limit)
```

It performs a search with restarts controlled by the resolution event [1] `counter` which counts events occurring during the search. Parameter `base` indicates the maximal number of events allowed in the first search tree. Once this limit is reached, a restart occurs and the search continues until `base``*``grow` events are done, and so on. After each restart, the limit number of events is increased by the geometric factor `grow`. `limit` states the maximum number of restarts.

and:

```
luby(Solver solver, int base, int grow, ICounter counter, int limit)
```

The Luby 's restart policy is an alternative to the geometric restart policy. It performs a search with restarts controlled by the number of resolution events [1] counted by `counter`. The maximum number of events allowed at a given restart iteration is given by base multiplied by the Las Vegas coefficient at this iteration. The sequence of these coefficients is defined recursively on its prefix subsequences: starting from the first prefix 1, the $(k+1)^t h$ prefix is the $k^t h$ prefix repeated `grow` times and immediately followed by coefficient `grow`^{k}.

- the first coefficients for `grow` =2: [1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,1,...]
- the first coefficients for `grow` =3 : [1, 1, 1, 3, 1, 1, 1, 3, 1, 1, 1, 3, 9,...]

8.6 Limiting the resolution

8.6.1 Built-in search limits

The exploration of the search tree can be limited in various ways. Some usual limits are provided in `SearchMonitorFactory`, or `SMF` for short:

- `limitTime` stops the search when the given time limit has been reached. This is the most common limit, as many applications have a limited available runtime.

Note: The potential search interruption occurs at the end of a propagation, i.e. it will not interrupt a propagation algorithm, so the overall runtime of the solver might exceed the time limit.

- `limitSolution` stops the search when the given solution limit has been reached.
- `limitNode` stops the search when the given search node limit has been reached.
- `limitFail` stops the search when the given fail limit has been reached.
- `limitBacktrack` stops the search when the given backtrack limit has been reached.

8.6.2 Custom search limits

You can decide to interrupt the search process whenever you want with one of the following instructions:

[1] Resolution events are: backtracks, fails, nodes, solutions, time or user-defined ones.

```
solver.getSearchLoop().reachLimit();
solver.getSearchLoop().interrupt(String message);
```

Both options will interrupt the search process but only the first one will inform the solver that the search stops because of a limit. In other words, calling

```
solver.hasReachedLimit()
```

will return false if the second option is used.

Going further

Large Neighborhood Search, *Explanations*.

CHAPTER 9

Resolution statistics

Resolution data are available thanks to the `Chatterbox` class, which outputs by default to `System.out`. It centralises widely used methods to have comprehensive feedback about the resolution process. There are two types of methods: those who need to be called **before** the resolution, with a prefix *show*, and those who need to called **after** the resolution, with a prefix *print*.

For instance, one can indicate to print the solutions all resolution long:

```
Chatterbox.showSolutions(solver);
solver.findAllSolutions();
```

Or to print the search statistics once the search ends:

```
solver.findSolution();
Chatterbox.printStatistics(solver);
```

On a call to `Chatterbox.printVersion()`, the following message will be printed:

```
** Choco 3.3.1 (2015-05) : Constraint Programming Solver, Copyleft (c) 2010-2015
```

On a call to `Chatterbox.printVersion()`, the following message will be printed:

```
- [ Search complete - [ No solution | {0} solution(s) found ]
  | Incomplete search - [ Limit reached | Unexpected interruption ] ].
   Solutions: {0}
[  Maximize = {1}  ]
[  Minimize = {2}  ]
   Building time : {3}s
   Resolution : {6}s
   Nodes: {7} ({7}/{6} n/s)
   Backtracks: {8}
   Fails: {9}
   Restarts: {10}
   Max depth: {11}
   Variables: {12}
   Constraints: {13}
```

Curly brackets *{instruction | }* indicate alternative instructions

Brackets *[instruction]* indicate an optional instruction.

If the search terminates, the message "Search complete" appears on the first line, followed with either the the number of solutions found or the message "No solution". `Maximize` –resp. `Minimize`– indicates the best known value before exiting of the objective value using a `ResolutionPolicy.MAXIMIZE` –resp. `ResolutionPolicy.MINIMIZE`- policy.

Curly braces *{value}* indicate search statistics:

0. number of solutions found
1. objective value in maximization
2. objective value in minimization
3. building time in second (from `new Solver()` to `findSolution()` or equivalent)
4. initialisation time in second (before initial propagation)
5. initial propagation time in second
6. resolution time in second (from `new Solver()` till now)
7. number of decision created, that is, nodes in the binary tree search
8. number of backtracks achieved
9. number of failures that occurred
10. number of restarts operated
11. maximum depth reached in the binary tree search
12. number of variables in the model
13. number of constraints in the model

If the resolution process reached a limit before ending *naturally*, the title of the message is set to :

```
- Incomplete search - Limit reached.
```

The body of the message remains the same. The message is formatted thanks to the `IMeasureRecorder` which is a *search monitor*.

On a call to `Chatterbox.showSolutions(solver)`, on each solution the following message will be printed:

```
{0} Solutions, [Maximize = {1}][Minimize = {2}], Resolution {6}s, {7} Nodes, \\
                                  {8} Backtracks, {9} Fails, {10} Restarts
```

followed by one line exposing the value of each decision variables (those involved in the search strategy).

On a call to `Chatterbox.showDecisions(solver)`, on each node of the search tree a message will be printed indicating which decision is applied. The message is prefixed by as many "." as nodes in the current branch of the search tree. A decision is prefixed with `[R]` and a refutation is prefixed by `[L]`.

```
..[L]x  ==  1 (0) //X = [0,5] Y = [0,6] ...
```

Warning: `Chatterbox.printDecisions(Solver solver)` prints the tree search during the resolution. Printing the decisions slows down the search process.

Part IV

Advanced usage

CHAPTER 10

Settings

A `Settings` object is attached to each `Solver`. It declares default behavior for various purposes: from general purpose (such as the welcome message), modelling purpose (such as enabling views) or solving purpose (such as the search binder).

The API is:

`String getWelcomeMessage()` Return the welcome message.

`Idem getIdempotencyStrategy()` Define how to react when a propagator is not ensured to be idempotent.

`boolean enableViews()` Set to 'true' to allow the creation of views in the `VariableFactory`. Creates new variables with channeling constraints otherwise.

`int getMaxDomSizeForEnumerated()` Define the maximum domain size threshold to force integer variable to be enumerated instead of bounded while calling `VariableFactory#integer(String, int, int, Solver)`.

`boolean enableTableSubstitution()` Set to true to replace intension constraints by extension constraints.

`int getMaxTupleSizeForSubstitution()` Define the maximum domain size threshold to replace intension constraints by extension constraints. Only checked when `enableTableSubstitution()` is set to true.

`boolean plugExplanationIn()` Set to true to plug explanation engine in.

`boolean enablePropagatorInExplanation()` Set to true to add propagators in explanations

`double getMCRPrecision()` Define the rounding precision for *multicost_regular*. MUST BE < 13 as java messes up the precisions starting from 10E-12 (34.0*0.05 == 1.70000000000005).

`double getMCRDecimalPrecision()` Defines the smallest used double for *multicost_regular*.

`short[] getFineEventPriority()` Defines, for fine events, for each priority, the queue in which a propagator of such a priority should be scheduled in.

`short[] getCoarseEventPriority()` Defines, for coarse events, for each priority, the queue in which a propagator of such a priority should be scheduled in

`ISearchBinder getSearchBinder()` Return the default *Search binder*.

`ICondition getEnvironmentHistorySimulationCondition()` Return the condition to satisfy when rebuilding history of backtrackable objects is needed.

`boolean warnUser()` Return true if one wants to be informed of warnings detected during modeling/solving (default value is false).

`boolean enableIncrementalityOnBoolSum(int nbvars)` Return true if the incrementality is enabled on boolean sum, based on the number of variables involved. Default condition is : nbvars > 10.

boolean outputWithANSIColors() If your terminal support ANSI colors (Windows terminals don't), you can set this to true and decisions and solutions will be output with colors.

boolean debugPropagation() When this setting returns true, a complete trace of the events is output. This can be quite big, though, and it slows down the overall process.

boolean cloneVariableArrayInPropagator() If this setting is set to true (default value), a clone of the input variable array is made in any propagator constructors. This prevents, for instance, wrong behavior when permutations occurred on the input array (e.g., sorting variables). Setting this to false may limit the memory consumption during modelling.

CHAPTER 11

Large Neighborhood Search (LNS)

Local search techniques are very effective to solve hard optimization problems. Most of them are, by nature, incomplete. In the context of constraint programming (CP) for optimization problems, one of the most well-known and widely used local search techniques is the Large Neighborhood Search (LNS) algorithm [1]. The basic idea is to iteratively relax a part of the problem, then to use constraint programming to evaluate and bound the new solution.

11.1 Principle

LNS is a two-phase algorithm which partially relaxes a given solution and repairs it. Given a solution as input, the relaxation phase builds a partial solution (or neighborhood) by choosing a set of variables to reset to their initial domain; The remaining ones are assigned to their value in the solution. This phase is directly inspired from the classical Local Search techniques. Even though there are various ways to repair the partial solution, we focus on the technique in which Constraint Programming is used to bound the objective variable and to assign a value to variables not yet instantiated. These two phases are repeated until the search stops (optimality proven or limit reached).

The `LNSFactory` provides pre-defined configurations. Here is the way to declare LNS to solve a problem:

```
LNSFactory.rlns(solver, ivars, 30, 20140909L, new FailCounter(solver, 100));
solver.findOptimalSolution(ResolutionPolicy.MINIMIZE, objective);
```

It declares a *random* LNS which, on a solution, computes a partial solution based on `ivars`. If no solution are found within 100 fails (`FailCounter(solver, 100)`), a restart is forced. Then, every `30` calls to this neighborhood, the number of fixed variables is randomly picked. `20140909L` is the seed for the `java.util.Random`.

The instruction `LNSFactory.rlns(solver, vars, level, seed, frcounter)` runs:

```
1       *
2       * @param solver    the solver
3       * @param vars      the pool of variables to choose from
```

The factory provides other LNS configurations together with built-in neighbors.

[1] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming, CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin Heidelberg, 1998.

11.2 Neighbors

While the implementation of LNS is straightforward, the main difficulty lies in the design of neighborhoods able to move the search further. Indeed, the balance between diversification (i.e., evaluating unexplored sub-tree) and intensification (i.e., exploring them exhaustively) should be well-distributed.

11.2.1 Generic neighbors

One drawback of LNS is that the relaxation process is quite often problem dependent. Some works have been dedicated to the selection of variables to relax through general concept not related to the class of the problem treated [5,24]. However, in conjunction with CP, only one generic approach, namely Propagation-Guided LNS [24], has been shown to be very competitive with dedicated ones on a variation of the Car Sequencing Problem. Nevertheless, such generic approaches have been evaluated on a single class of problem and need to be thoroughly parametrized at the instance level, which may be a tedious task to do. It must, in a way, automatically detect the problem structure in order to be efficient.

11.2.2 Combining neighborhoods

There are two ways to combine neighbors.

Sequential

Declare an instance of `SequenceNeighborhood(n1, n2, ..., nm)`. Each neighbor ni is applied in a sequence until one of them leads to a solution. At step k, the $(k \mod m)^{th}$ neighbor is selected. The sequence stops if at least one of the neighbor is complete.

Adaptive

Declare an instance of `AdaptiveNeighborhood(1L, n1, n2, ..., nm)`. At the beginning a weight of 1 at assigned to each neighbor ni. Then, if a neighbor leads to solution, its weight w_i is increased by 1. Any time a partial solution has to be computed, a value `W` between 1 and $w_1 + w_2 + ... + w_n$ is randomly picked (`1L` is the seed). Then the weight of each neighbor is subtracted from `W`, as soon as `W`≤ 0, the corresponding neighbor is selected. For instance, let's consider three neighbors n1, n2 and n3, their respective weights w1=2, w2=4, w3=1. `W` = 3 is randomly picked between 1 and 7. Then, the weight of n1 is subtracted, `W``2-=1; the weight of n2 is subtracted, ``W`-4 = -3, `W` is less than 0 and n2 is selected.

11.2.3 Defining its own neighborhoods

One can define its own neighbor by extending the abstract class `INeighbor`. It forces to implements the following methods:

Method	Definition
`void recordSolution()`	Action to perform on a solution (typicallu, storing the current variables' value).
`Decision fixSomeVariables()`	Fix some variables to their value in the last solution, computing a partial solution and returns it as a decision.
`void restrictLess()`	Relax the number of variables fixed. Called when no solution was found during a LNS run (trapped into a local optimum).
`boolean isSearchComplete()`	Indicates whether the neighbor is complete, that is, can end.

11.3 Restarts

A generic and common way to reinforce diversification of LNS is to introduce restart during the search process. This technique has proven to be very flexible and to be easily integrated within standard backtracking procedures [2].

11.4 Walking

A complementary technique that appear to be efficient in practice is named *Walking* and consists in accepting equivalent intermediate solutions in a search iteration instead of requiring a strictly better one. This can be achieved by defining an `ObjectiveManager` like this:

```
solver.set(new ObjectiveManager(objective, ResolutionPolicy.MAXIMIZE, false));
```

Where the last parameter, named `strict` must be set to false to accept equivalent intermediate solutions.

Other optimization policies may be encoded by using either search monitors or a custom `ObjectiveManager`.

[2] Laurent Perron. Fast restart policies and large neighborhood search. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming at CP 2003*, volume 2833 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003.

CHAPTER 12

Multi-thread resolution

Choco 3 provides a simple way to use several thread to treat a problem. This is achieved by using Java8 lambdas. The main idea of that driver is to solve the *same* problem with various search strategies, and to share few possible information.

The first step is to declare a method populates (adding variables and constraints) a solver given in parameter. The *n* solvers should be passed to that method.

Before running the resolution, a call to the following method is required to synchronize interruption and, when dealing with optimization problems, to share data relative to the objective variable(s) between solvers:

```
SearchMonitorFactory.prepareForParallelResolution(List<Solver> solvers)
```

Make sure, if dealing with an optimization problem, that the objective variable is eagerly declared with `solver.setObjectives(cost);`.

Finally, the expected ways to solve a problem using mutlple solvers is:

```
int n =4; // number of solvers to use
List<Solver> solvers = new ArrayList<>();
for(int i = 0 ; i < n; i++){
   Solver solver = new Solver();
   solvers.add(solver);
   readModel(solver); // a dedicated method that declares variables and constraints
   // the search should also be declared within that method
}
SMF.prepareForParallelResolution(solvers);
solvers.parallelStream().forEach(s -> {
    s.findOptimalSolution(ResolutionPolicy.MINIMIZE);
});
```

When dealing with multithreading resolution, very few data is shared between threads: when a solver ends, it communicates an interruption instruction to the others and the best known bound of the objective variable(s) is shared among solver. This enables to explore the search space in various way, setting distinct search strategy and/or search loop (this should be done in the dedicated method which builds the model, though) This also enables to solve multiple modelling of the same problemi (or even distinct problems), as long as the resolution policy remains the same.

CHAPTER 13

Explanations

Choco 3 natively support explanations [1]. However, no explanation engine is plugged-in by default.

13.1 Principle

Nogoods and explanations have long been used in various paradigms for improving search. An explanation records some sufficient information to justify an inference made by the solver (domain reduction, contradiction, etc.). It is made of a subset of the original propagators of the problem and a subset of decisions applied during search. Explanations represent the logical chain of inferences made by the solver during propagation in an efficient and usable manner. In a way, they provide some kind of a trace of the behavior of the solver as any operation needs to be explained.

Explanations have been successfully used for improving constraint programming search process. Both complete (as the mac-dbt algorithm) and incomplete (as the decision-repair algorithm) techniques have been proposed. Those techniques follow a similar pattern: learning from failures by recording each domain modification with its associated explanation (provided by the solver) and taking advantage of the information gathered to be able to react upon failure by directly pointing to relevant decisions to be undone. Complete techniques follow a most-recent based pattern while incomplete technique design heuristics to be used to focus on decisions more prone to allow a fast recovery upon failure.

The current explanation engine is coded to be *Asynchronous, Reverse, Low-intrusive and Lazy*:

Asynchronous: Explanations are not computed during the propagation.

Reverse: Explanations are computed in a bottom-up way, from the conflict to the first event generated, *keeping* only relevant events to compute the explanation of the conflict.

Low-intrusive: Basically, propagators need to implement only one method to furnish a convenient explanation schema.

Lazy: Explanations are computed on request.

To do so, all events are stored during the descent to a conflict/solution, and are then evaluated and kept if relevant, to get the explanation.

13.2 In practice

Consider the following example:

[1] Narendra Jussien. The versatility of using explanations within constraint programming. Technical Report 03-04-INFO, 2003.

```
        Solver solver = new Solver();
        BoolVar[] bvars = VF.boolArray("B", 4, solver);
        solver.post(ICF.arithm(bvars[2], "=", bvars[3]));
        solver.post(ICF.arithm(bvars[2], "!=", bvars[3]));
        solver.set(ISF.lexico_LB(bvars));
        solver.findAllSolutions();
```

The problem has no solution since the two constraints cannot be satisfied together. A naive strategy such as `ISF.lexico_LB(bvars)` (which selects the variables in lexicographical order) will detect lately and many times the failure. By plugging-in an explanation engine, on each failure, the reasons of the conflict will be explained.

```
        ExplanationFactory.CBJ.plugin(solver, false, false);
```

The explanation engine records *deductions* and *causes* in order to compute explanations. In that small example, when an explanation engine is plugged-in, the two first failures will enable to conclude that the problem has no solution. Only three nodes are created to close the search, seven are required without explanations.

Note: Only unary, binary, ternary and limited number of nary propagators over integer variables have a dedicated explanation algorithm. Although global constraints over integer variables are compatible with explanations, they should be either accurately explained or reformulated to fully benefit from explanations.

13.2.1 Cause

A cause implements `ICause` and must defined the `boolean why(RuleStore ruleStore, IntVar var, IEventType evt, int value)` method. Such a method add new *event filtering* rules to the ruleStore in parameter in order to *filter* relevant events among all generated during the search. Every time a variable is modified, the cause is specified in order to compute explanations afterwards. For instance, when a propagator updates the bound of an integer variable, the cause is the propagator itself. So do decisions, objective manager, etc.

13.2.2 Computing explanations

When a contradiction occurs during propagation, it can only be thrown by:

- a propagator which detects unsatisfiability, based on the current domain of its variables;
- or a variable whom domain became empty.

Consequently, in addition to causes, variables can also explain the current state of their domain. Computing the explanation of a failure consists in going up in the stack of all events generated in the current branch of the search tree and filtering the one relative to the conflict. The entry point is either a the unsatisfiabable propagator or the empty variable.

Note: Explanations can be computed without failure. The entry point is a variable, and only removed values can be explained.

Each propagator embeds its own explanation algorithm which relies on the relation it defines over variables.

Warning: Even if a naive (and weak) explanation algorithm could be provided by all constraints, we made the choice to throw an *SolverException* whenever a propagator does not defined its own explanation algorithm. This is restrictive, but almost all non-global constraints support explanation, which enables reformulation. The missing explanation schemas will be integrated all needs long.

For instance, here is the algorithm of `PropGreaterOrEqualX_YC` ($x \geq y + c$, `x` and `y` are integer variables, `c` is a constant):

```
1         boolean newrules = ruleStore.addPropagatorActivationRule(this);
2         if (var.equals(x)) {
3             newrules |= ruleStore.addLowerBoundRule(y);
4         } else if (var.equals(y)) {
5             newrules |= ruleStore.addUpperBoundRule(x);
6         } else {
7             newrules |= super.why(ruleStore, var, evt, value);
8         }
9         return newrules;
10    }
```

The first lines indicates that the deduction is due to the application of the propagator (l.2), maybe through reification. Then, depending on the variable touched by the deduction, either the lower bound of `y` (l.4) or the upper bound of `x` (l.6) explains the deduction. Indeed, such a propagator only updates lower bound of `y` based on the upper bound of `x` and *vice versa*.

Let consider that the deduction involves `x` and is explained by the lower bound of `y`. The lower bound `y` needs to be explained. A new rule is added to the ruleStore to specify that events on the lower bound of `y` needs to be kept during the event stack analyse (only events generated before the current are relevant). When such events are found, the ruleStore can be updated, until the first event is analyzed.

The results is a set of branching decisions, and a set a propagators, which applied altogether leads the conflict and thus, explained it.

13.3 Explanations for the system

Explanations for the system, which try to reduce the search space, differ from the ones giving feedback to a user about the unsatisfiability of its model. Both rely on the capacity of the explanation engine to motivate a failure, during the search form system explanations and once the search is complete for user ones.

Important: Most of the time, explanations are raw and need to be processed to be easily interpreted by users.

13.3.1 Conflict-based backjumping

When Conflict-based Backjumping (CBJ) is plugged-in, the search is hacked in the following way. On a failure, explanations are retrieved. From all left branch decisions explaining the failure, the last taken, *return decision*, is stored to jump back to it. Decisions from the current one to the return decision (excluded) are erased. Then, the return decision is refuted and the search goes on. If the explanation is made of no left branch decision, the problem is proven to have no solution and search stops.

Factory: `solver.explanations.ExplanationFactory`

API:

```
CBJ.plugin(Solver solver, boolean nogoodsOn, boolean userFeedbackOn)
```

- *solver*: the solver to explain.
- *nogoodsOn*: set to *true* to extract nogood from each conflict,. Extracting nogoods slows down the overall resolution but can reduce the search space.

- *userFeedbackOn*: set to *true* to store the very last explanation of the search (recommended value: *false*).

13.3.2 Dynamic backtracking

This strategy, Dynamic backtracking (DBT) corrects a lack of deduction of Conflict-based backjumping. On a failure, explanations are retrieved. From all left branch decisions explaining the failure, the last taken, *return decision*, is stored to jump back to it. Decisions from the current one to the return decision (excluded) are maintained, only the return decision is refuted and the search goes on. If the explanation is made of no left branch decision, the problem is proven to have no solution and search stops.

Factory: `solver.explanations.ExplanationFactory`

API:

```
DBT.plugin(Solver solver, boolean nogoodsOn, boolean userFeedbackOn)
```

- *solver*: the solver to explain.
- *nogoodsOn*: set to *true* to extract nogood from each conflict,. Extracting nogoods slows down the overall resolution but can reduce the search space.
- *userFeedbackOn*: set to *true* to store the very last explanation of the search (recommended value: *false*).

13.4 Explanations for the end-user

Explaining the last failure of a complete search without solution provides information about the reasons why a problem has no solution. For the moment, there is no simplified way to get such explanations. CBJ and DBT enable retrieving an explanation of the last conflict.

```
// .. problem definition ..
// First manually plug CBJ, or DBT
ExplanationEngine ee = new ExplanationEngine(solver, userFeedbackOn);
ConflictBackJumping cbj = new ConflictBackJumping(ee, solver, nogoodsOn);
solver.plugMonitor(cbj);
if(!solver.findSolution()){
    // If the problem has no solution, the end-user explanation can be retrieved
    System.out.println(cbj.getLastExplanation());
}
```

Incomplete search leads to incomplete explanations: as far as at least one decision is part of the explanation, there is no guarantee the failure does not come from that decision. On the other hand, when there is no decision, the explanation is complete.

CHAPTER 14

Search loop

The search loop whichs drives the search is a freely-adapted version PLM [1]. PLM stands for: Propagate, Learn and Move. Indeed, the search loop is composed of three parts, each of them with a specific goal.

- Propagate: it aims at propagating information throughout the constraint network when a decision is made,
- Learn: it aims at ensuring that the search mechanism will avoid (as much as possible) to get back to states that have been explored and proved to be solution-less,
- Move: it aims at, unlike the former ones, not pruning the search space but rather exploring it.

Any component can be freely implemented and attached to the search loop in order to customize its behavior. There exists some pre-defined *Move* and *Learn* implementations, avaiable in *Search loop factory*.

Move:

dfs, *lds*, *dds*, *hbfs*, *seq*, *restart*, *restartOnSolutions*, *lns*.

Learn:

learnCBJ, *learnDBT*,

One can also define its own *Move* or *Learn* implementation, more details are given in *Implementing a search loop component*.

[1] Narendra Jussien and Olivier Lhomme. Unifying search algorithms for CSP. Technical report 02-3-INFO, EMN.

CHAPTER 15

Search monitor

15.1 Principle

A search monitor is an observer of the search loop. It gives user access before and after executing each main step of the search loop:

- *initialize*: when the search loop starts and the initial propagation is run,
- *open node*: when a decision is computed,
- *down branch*: on going down in the tree search applying or refuting a decision,
- *up branch*: on going up in the tree search to reconsider a decision,
- *solution*: when a solution is got,
- *restart search*: when the search is restarted to a previous node, commonly the root node,
- *close*: when the search loop ends,
- *contradiction*: on a failure,

With the accurate search monitor, one can easily observe with the search loop, from pretty printing of a solution to learning nogoods from restart, or many other actions.

The interfaces to implement are:

- `IMonitorInitialize`,
- `IMonitorOpenNode`,
- `IMonitorDownBranch`,
- `IMonitorUpBranch`,
- `IMonitorSolution`,
- `IMonitorRestart`,
- `IMonitorContradiction`,
- `IMonitorClose`.

Most of them gives the opportunity to do something before and after a step. The other ones are called after a step.

For instance, `NogoodStoreFromRestarts` monitors restarts. Before a restart is done, the nogoods are extracted from the current decision path; after the restart has been done, the newly created nogoods are added and the nogoods are propagated. Thus, the framework is almost not intrusive.

```
*/
    @Override
    public void beforeRestart() {
        extractNogoodFromPath();
    private void extractNogoodFromPath() {
        int d = (int) png.getSolver().getMeasures().getNodeCount();
```

Available search monitors: *Search Monitors*.

Important: A search monitor should not interact with the search loop (forcing restart and interrupting the search, for instance). This is the goal of the Move component of a search loop *Search loop*.

CHAPTER 16

Defining its own search strategy

One key component of the resolution is the exploration of the search space induced by the domains and constraints. It happens that built-in search strategies are not enough to tackle the problem. Or one may want to define its own strategy. This can be done in three steps: selecting the variable, selecting the value, then making a decision.

The following instructions are based on IntVar, but can be easily adapted to other types of variables.

16.1 Selecting the variable

An implementation of the `VariableSelector<V extends Variable>` interface is needed. A variable selector specifies which variable should be selected at a fix point. It is based specifications (ex: smallest domain, most constrained, etc.). Although it is not required, the selected variable should not be already instantiated to a singleton. This interface forces to define only one method:

```
V getVariable(V[] variables)
```

One variable has to be selected from `variables` to create a decision on. If no valid variable exists, the method is expected to return `null`.

An implementation of the `VariableEvaluator<V extends Variable>` is strongly recommended. It enables breaking ties. It forces to define only one method:

```
double evaluate(V variable)
```

An evaluation of the given variable is done wrt the evaluator. The variable with the **smallest** value will then be selected.

Here is the code of the `FirstFail` variable selector which selects first the variable with the smallest domain.

```
 */
public class FirstFail implements VariableSelector<IntVar>, VariableEvaluator<IntVar> {


    @Override
    public IntVar getVariable(IntVar[] variables) {
        int small_idx = -1;
        int small_dsize = Integer.MAX_VALUE;
        for (int idx = 0; idx < variables.length; idx++) {
            int dsize = variables[idx].getDomainSize();
            if (dsize > 1 && dsize < small_dsize) {
                small_dsize = dsize;
                small_idx = idx;
            }
```

```
15          }
16          return small_idx > -1 ? variables[small_idx] : null;
17      }
18
19      @Override
20      public double evaluate(IntVar variable) {
21          return variable.getDomainSize();
22      }
```

There is a distinction between *VariableSelector* and *VariableEvaluator*. On the one hand, a *VariableSelector* breaks ties lexicographically, that is, the first variable in the input array which respects the specification is returned.

```
new IntStrategy(variables,
                new FirstFail(),
                new IntDomainMin(),
                DecisionOperator.int_eq);
```

On the other hand, a *VariableEvaluator* selects all variables which respect the specifications and let another *VariableEvaluator* breaks ties, if any, or acts like a *VariableSelector*.

```
new IntStrategy(variables,
                new VariableSelectorWithTies(new FirstFail(), new Largest()),
                new IntDomainMin(),
                DecisionOperator.int_eq);
```

Let's consider the following array of variables as input *{X,Y,Z}* where *X=[0,3], Y= [0,4]* and *Z=[1,4]*. Applying the first strategy declared will return *X*. Applying the second one will return *Z*: *X* and *Z* are batter than *Y* but equivalent compared to *FirstFail* but *Z* is better than *X* compared to *Largest*.

16.2 Selecting the value

The value to be selected must belong to the variable domain.

For `IntVar` the interface `IntValueSelector` is required. It imposes one method:

```
int selectValue(IntVar var)
```

Return the value to constrain `var` with.

Important: A value selector must consider the type of domain of the selected variable. Indeed, a value selector does not store the previous tries (unkike an iterator) and it may happen that, for bounded variable, the refutation of a decision has no effect and a value is selected twice or more. For example, consider *IntDomainMiddle* and a bounded variable.

16.3 Making a decision

A decision is made of a variable, an decision operator and a value. The decision operator should be selected in `DecisionOperator` among:

```
int_eq
```

For `IntVar`, represents an instantiation, $X = 3$. The refutation of the decision will be a value removal.

`int_neq`

For `IntVar`, represents a value removal, $X \neq 3$. The refutation of the decision will be an instantiation.

`int_split`

For `IntVar`, represents an upper bound modification, $X \leq 3$. The refutation of the decision will be a lower bound modification.

`int_reverse_split`

For `IntVar`, represents a lower bound modification, $X \geq 3$. The refutation of the decision will be an upper bound modification.

`set_force`

For `SetVar`, represents a kernel addition, $3 \in S$. The refutation of the decision will be an envelop removal.

`set_remove`

For `SetVar`, represents an envelop removal, $3 \notin S$. The refutation of the decision will be a kernel addition.

Attention: A particular attention should be made while using `IntVar` s and their type of domain. Indeed, bounded variables does not support making holes in their domain. Thus, removing a value which is not a current bound will be missed, and can lead to an infinite loop.

One can define its own operator by extending `DecisionOperator`.

`void apply(V var, int value, ICause cause)`

Operations to execute when the decision is applied (left branch). It can throw an `ContradictionException` if the application is not possible.

`void unapply(V var, int value, ICause cause)`

Operations to execute when the decision is refuted (right branch). It can throw an `ContradictionException` if the application is not possible.

`DecisionOperator opposite()`

Opposite of the decision operator. *Currently useless*.

`String toString()`

A pretty print of the decision, for logging.

Most of the time, extending `AbstractStrategy` is not necessary. Using specific strategy dedicated to a type of variable, such as `IntStrategy` is enough. The one above has an alternate constructor:

```
public IntStrategy(IntVar[] scope,
                   VariableSelector<IntVar> varSelector,
                   IntValueSelector valSelector,
                   DecisionOperator<IntVar> decOperator) {...}
```

And defining your own strategy is really crucial, start by copying/pasting an existing one. Indeed, decisions are stored in pool managers to avoid creating too many decision objects, and thus garbage collecting too often.

CHAPTER 17

Defining its own constraint

In Choco-3, constraints is basically a list of filtering algorithms, called *propagators*. A propagator is a function from domains to domains which removes impossible values from variable domains.

17.1 Structure of a Propagator

A propagator needs to extends the `Propagator` abstract class. Then, a constructor and some methods have to be implemented:

`super(...)`

> a call to `super()` is mandatory. The list of variables (which determines the index of the variable in the propagator) and the priority (for the propagation engine) are required. An optional boolean (`true` is the default value) can be set to `false` to avoid reacting on fine events (see item `void propagate(int vIdx, int mask)`). More precisely, if set to `false`, the propagator will only be informed of a modification of, at least, one of its variables, without knowing specifically which one(s) and what modifications occurred.

Important: The array of variables given in parameter of a `Propagator` constructor is not cloned but referenced. That is, if a permutation occurs in the array of variables, all propagators referencing the array will be incorrect.

`ESat isEntailed()`

> This method is mandatory for reification. It checks whether the propagator will be always satisfied (`ESat.TRUE`), never satisfied (`ESat.FALSE`) or undefined (`ESat.UNDEFINED`) according to the current state of its domain variables and/or its internal structure. By default, it should consider the case where all variables are instantiated. For instance, $A \neq B$ will always be satisfied when $A={0,1,2}$ and $B = \{4, 5\}$. For instance, $A = B$ will never be satisfied when $A = \{0, 1, 2\}$ and $B = \{4, 5\}$. For instance, entailment of $A \neq B$ cannot be defined when $A = \{0, 1, 2\}$ and $B = \{1, 2, 3\}$.

This method is also called to check solutions when assertions are enabled, i.e. when the *-ea* JVM option is used.

`void propagate(int evtmask)`

> This method applies the global filtering algorithm of the propagator, that is, from *scratch*. It is called once during initial propagation and then on a call to `forcePropagate(EventType)`. There are two available types of event this method can receive: `EventType.FULL\_PROPAGATION` and `EventType.CUSTOM\_PROPAGATION`. The latter is propagator-dependent and should be managed by the developer when incrementality is enabled. Note that the `forcePropagate()` method will call `propagate(int)` when the propagator does not have any pending events. In other words, it is called once and for all, after many domain modifications.

`void propagate(int vIdx, int mask)`

This method is the main entry point to the propagator during propagation. When the $vIdx^{th}$ variable of the propagator is modified, data relative to the modification is stored for a future execution of the propagator. Then, when the propagation engine has to execute the propagator, a call to this method is done with the data relative to the variable and its modifications. One can delegate filtering algorithm to `propagate(int)` with a call to `forcePropagate()` (see item `void propagate(int evtmask)`). However, developers have to be aware that a propagator will not be informed of a modification it has generated itself. That's why a propagator has to be idempotent (see Section~nameref{properties}) or being aware not to be.

Note that, when conditions enable it, a call to `setPassive()` will deactivate the propagator temporary, during the exploration of the sub search space. When the conditions are not met anymore, the propagator is activated again (i.e. on backtrack).

`int getPropagationConditions(int vIdx)`

This method returns the specific mask indicating the variable events on which the propagator reacts for the $vIdx^{th}$ variable. This method is related to `propagate(int, int)`: a wrong mask prevents the propagator from being informed of an event occurring on a variable. Event masks are not nested and all event masks have to be defined.

17.2 Properties

We distinguish two kinds of propagators:

Basis propagators, that ensure constraints to be satisfied.

Redundant (or *Implied*) propagators that come in addition to some basis propagators, in order to get a stronger filtering.

A basis propagator should be idempotent [1] . A redundant propagator does not have to be idempotent:

Some propagators cannot be idempotent because they are not even monotonic [2] (Lagrangian relaxation, use of randomness, etc.),

Forcing to reach the fix point may decrease performances.

Important: A redundant propagator can directly return `ESat.TRUE` in the body of the `isEntailed()` method. Indeed, it comes in addition to basis propagators that will already ensure constraint satisfaction.

17.3 How to make a propagator idempotent?

Trying to make a propagator idempotent directly may not be straightforward. We provide three implementation possibilities.

The *coarse* option:

the propagator will perform its fix point by itself. The propagator does not react to fine events. The coarse filtering algorithm should be surrounded like this:

[1] **idempotent**: calling a propagator twice has no effect, i.e. calling it with its output domains returns its output domains. In that case, it has reached a fix point.

[2] **monotonic**: calling a propagator with two input domains A and B for which $A \subseteq B$ returns two output domains A' and B' for which $A' \subseteq B'$.

```
long size;
do{
  size = 0;
  for(IntVar v:vars){
    size+=v.getDomSize();
  }
  // really update domain variables here
  for(IntVar v:vars){
    size-=v.getDomSize();
  }
}while(size>0);
```

Important: Domain variable modifier returns a boolean valued to `true` if the domain variable has been modified.

Important: In the case of `SetVar` or `GraphVar`, replace `getDomSize()` by `getEnvSize()-getKerSize()`.

The *decomposed* option:

> Split the original propagator into many propagators so that the fix point is performed through the propagation engine. For instance, a channeling propagator $A \Leftrightarrow B$ can be decomposed into two propagators $A \Rightarrow B$ and $B \Rightarrow A$. The propagators can (but does not have to) react on fine events.

The *lazy* option:

> (To be avoided has much as possible) simply post the propagator twice. Thus, the fix point is performed through the propagation engine.

CHAPTER 18

Implementing a search loop component

A search loop is made of three components, each of them dealing with a specific aspect of the search. Even if many *Move* and *Learn* implementation are already provided, it may be relevant to define its own component.

Note: The *Propagate* component is less prone to be modified, it will not be described here. However, its interface is minimalist and can be easily implemented. A look to *org.chocosolver.solver.search.loop.PropagateBasic.java* is a good starting point.

The two components can be easily set in the *Solver* search loop:

`void setMove(Move m)` The current *Move* component is replaced by *m*.

`Move getMove()` The current *Move* component is returned.

void setLearn(Learn l) and *Learn getLearn()* are also avaiable.

Having access to the current *Move* (resp. *Learn*) component can be useful to combined it with another one. For instance, the *MoveLNS* is activated on a solution and creates a partial solution. It needs another *Move* to find the first solution and to complete the partial solution.

18.1 Move

Here is the API of *Move*:

`boolean extend(SearchLoop searchLoop)` Perform a move when the CSP associated to the current node of the search space is not proven to be not consistent. It returns *true* if an extension can be done, *false* when no more extension is possible. It has to maintain the correctness of the reversibility of the action by pushing a backup world when needed. An extension is commonly based on a decision, which may be made on one or many variables. If a decision is created (thanks to the search strategy), it has to be linked to the previous one.

`boolean repair(SearchLoop searchLoop)` Perform a move when the CSP associated to the current node of the search space is proven to be not consistent. It returns *true* if a reparation can be done, *false* when no more reparation is possible. It has to backtracking backup worlds when needed, and unlinked useless decisions. The depth and number of backtracks have to be updated too, and "up branch" search monitors of the search loop have to called (be careful, when many *Move* are combined).

`Move getChildMove()` It returns the child *Move* or *null*.

`void setChildMove(Move aMove)` It defined the child *Move* and erases the previously defined one, if any.

`boolean init()` Called before the search starts, it should initialize the search strategy, if any, and its child *Move*. It should return *false* if something goes wrong (the problem has trivially no solution), *true* otherwise.

`AbstractStrategy<V> getStrategy()` It returns the search strategy in use, which may be *null* if none has been defined.

`void setStrategy(AbstractStrategy<V> aStrategy)` It defines a search strategy and erases the previously defined one, that is, a service which computes and returns decisions.

`org.chocosolver.solver.search.loop.MoveBinaryDFS.java` is good starting point to see how a *Move* is implemented. It defines a Depth-First Search with binary decisions.

18.2 Learn

The aim of the component is to make sure that the search mechanism will avoid (as much as possible) to get back to states that have been explored and proved to be solution-less. Here is the API of *Learn*

`void record(SearchLoop searchLoop)` It validates and records a new piece of knowledge, that is, the current position is a dead-end. This is alwasy called *before* calling *Move.repair(SearchLoop)*.

`void forget(SearchLoop searchLoop)` It forgets some pieces of knowledge. This is alwasy called *after* calling *Move.repair(SearchLoop)*.

`org.chocosolver.solver.search.loop.LearnCBJ` is good, yet not trivial, example of *Learn*.

CHAPTER 19

Ibex

> "IBEX is a C++ library for constraint processing over real numbers.
>
> It provides reliable algorithms for handling non-linear constraints. In particular, round off errors are also taken into account. It is based on interval arithmetic and affine arithmetic." – http://www.ibex-lib.org/

To manage continuous constraints with Choco, an interface with Ibex has been done. It needs Ibex to be installed on your system. Then, simply declare the following VM options:

```
-Djava.library.path=/path/to/Ibex/lib
```

The path */path/to/Ibex/lib* points to the *lib* directory of the Ibex installation directory.

19.1 Installing Ibex

See the installation instructions of Ibex to complied Ibex on your system. More specially, take a look at Installation as a dynamic library and do not forget to add the `--with-java-package=org.chocosolver.solver.constraints.real` configuration option.

Once the installation is completed, the JVM needs to know where Ibex is installed to fully benefit from the Choco-Ibex bridge and declare real variables and constraints.

Part V

Elements of Choco

CHAPTER 20

Constraints over integer variables

20.1 absolute

The `absolute` constraint involves two variables *VAR1* and *VAR2*. It ensures that *VAR1* = | *VAR2* | .

API:

```
Constraint absolute(IntVar VAR1, IntVar VAR2)
```

Example

```
1         Solver solver = new Solver();
2         IntVar X = VF.enumerated("X", 0, 2, solver);
3         IntVar Y = VF.enumerated("X", -6, 1, solver);
4         solver.post(ICF.absolute(X, Y));
5         solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 0, Y = 0*
- *X = 1, Y = -1*
- *X = 1, Y = 1*
- *X = 2, Y = -2*

20.2 alldifferent

The *alldifferent* constraints involves two or more integer variables *VARS* and holds that all variables from *VARS* take a different value. A signature offers the possibility to specify the filtering algorithm to use:

- `"BC"`: filters on bounds only, based on [LopezOrtizQTvB03].
- `"AC"`: filters on the entire domain of the variables, based on [Regin94]. It runs in *O(m.n)*

worst case time for the initial propagation. The average runtime of further propagations is *O(n+m)*. - `"DEFAULT"`: uses `"BC"` plus a probabilistic `"AC"` propagator to get a compromise between `"BC"` and `"AC"`.

See also: alldifferent in the Global Constraint Catalog.

Implementation based on: [Regin94], [LopezOrtizQTvB03].

API:

```
Constraint alldifferent(IntVar[] VARS)
Constraint alldifferent(IntVar[] VARS, String CONSISTENCY)
```

Example

```
        Solver solver = new Solver();
        IntVar W = VF.enumerated("W", 0, 1, solver);
        IntVar X = VF.enumerated("X", -1, 2, solver);
        IntVar Y = VF.enumerated("Y", 2, 4, solver);
        IntVar Z = VF.enumerated("Z", 5, 7, solver);
        solver.post(ICF.alldifferent(new IntVar[]{W, X, Y, Z}));
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *X = -1, Y = 2, Z = 5, W = 1*
- *X = 1, Y = 2, Z = 7, W = 0*
- *X = 2, Y = 3, Z = 5, W = 0*
- *X = 2, Y = 4, Z = 7, W = 1*

20.3 alldifferent_conditionnal

The *alldifferent_conditionnal* constraint is a variation of the *alldifferent* constraint. It holds the *alldifferent* constraint on the subset of variables *VARS* which satisfies the given condition *CONDITION*.

A simple example is the *alldifferent_except_0* variation of the *alldifferent* constraint.

API:

```
Constraint alldifferent_conditionnal(IntVar[] VARS, Condition CONDITION)
Constraint alldifferent_conditionnal(IntVar[] VARS, Condition CONDITION, boolean AC)
```

One can force the *AC* algorithm to be used by calling the second signature.

Example

```
        Solver solver = new Solver();
        IntVar[] XS = VF.enumeratedArray("XS", 5, 0, 3, solver);
        solver.post(ICF.alldifferent_conditionnal(XS,
                x -> !x.contains(1) && !x.contains(3)));
        Chatterbox.showSolutions(solver);
        solver.findAllSolutions();
        solver.findAllSolutions();
```

The condition in the example states that the values *1* and *3* can appear more than once, unlike other values.

Some solutions of the problem are :

- *XS[0] = 0, XS[1] = 1, XS[2] = 1, XS[3] = 1, XS[4] = 1*

- *XS[0] = 0, XS[1] = 1, XS[2] = 2, XS[3] = 1, XS[4] = 1*
- *XS[0] = 1, XS[1] = 2, XS[2] = 1, XS[3] = 1, XS[4] = 1*
- *XS[0] = 0, XS[1] = 1, XS[2] = 2, XS[3] = 3, XS[4] = 3*

20.4 alldifferent_except_0

The *alldifferent_except_0* involves an array of variables *VARS*. It ensures that all variables from *VAR* take a distinct value or 0, that is, all values but 0 can't appear more than once.

See also: alldifferent_except_0 in the Global Constraint Catalog.

API:

```
Constraint alldifferent_except_0(IntVar[] VARS)
```

Example

```
1         Solver solver = new Solver();
2         IntVar[] XS = VF.enumeratedArray("XS", 4, 0, 2, solver);
3         solver.post(ICF.alldifferent_except_0(XS));
4         solver.findAllSolutions();
```

Some solutions of the problem are :

- *XS[0] = 0, XS[1] = 0, XS[2] = 0, XS[3] = 0*
- *XS[0] = 0, XS[1] = 1, XS[2] = 2, XS[3] = 0*
- *XS[0] = 0, XS[1] = 2, XS[2] = 0, XS[3] = 0*
- *XS[0] = 2, XS[1] = 1, XS[2] = 0, XS[3] = 0*

20.5 among

The *among* constraint involves:

- an integer variable *NVAR*,
- an array of integer variables *VARIABLES* and
- an array of integers.

It holds that *NVAR* is the number of variables of the collection *VARIABLES* that take their value in *VALUES*.

See also: among in the Global Constraint Catalog.

Implementation based on: [BessiereHH+05].

API:

```
Constraint among(IntVar NVAR, IntVar[] VARS, int[] VALUES)
```

Example

```
        Solver solver = new Solver();
        IntVar N = VF.enumerated("N", 2, 3, solver);
        IntVar[] XS = VF.enumeratedArray("XS", 4, 0, 6, solver);
        solver.post(ICF.among(N, XS, new int[]{1, 2, 3}));
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *N = 2, XS[0] = 0, XS[1] = 0, XS[2] = 1, XS[3] = 1*
- *N = 2, XS[0] = 0, XS[1] = 1, XS[2] = 3, XS[3] = 6*
- *N = 3, XS[0] = 1, XS[1] = 1, XS[2] = 2, XS[3] = 4*
- *N = 3, XS[0] = 3, XS[1] = 2, XS[2] = 1, XS[3] = 0*

20.6 arithm

The constraint *arithm* involves either:

- a integer variable *VAR*, an operator *OP* and a constant *CST*. It holds *VAR OP CSTE*, where *CSTE* must be chosen in {"=", "!=", ">","<",">=","<="}.
- or two variables *VAR1* and *VAR2* and an operator *OP*. It ensures that *VAR1 OP VAR2*, where *OP* must be chosen in {"=", "!=", ">","<",">=","<="} .
- or two variables *VAR1* and *VAR2*, two operators *OP1* and *OP2* and an constant *CSTE*. The operators must be different, taken from {"=", "!=", ">","<",">=","<="} or {"+", "-"}, the constarint ensures that *VAR1 OP1 VAR2 OP2 CSTE*.

API:

```
Constraint arithm(IntVar VAR, String OP, int CSTE)
Constraint arithm(IntVar VAR1, String OP, IntVar VAR2)
Constraint arithm(IntVar VAR1, String OP1, IntVar VAR2, String OP2, int CSTE)
```

Example 1

```
        Solver solver = new Solver();
        IntVar X = VF.enumerated("X", 1, 4, solver);
        solver.post(ICF.arithm(X, ">", 2));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 3*
- *X = 4*

Example 2

```
        Solver solver = new Solver();
        IntVar X = VF.enumerated("X", 0, 2, solver);
        IntVar Y = VF.enumerated("X", -6, 1, solver);
```

```
        solver.post(ICF.arithm(X, "<=", Y, "+", 1));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 0, Y = -1*
- *X = 0, Y = 0*
- *X = 0, Y = 1*
- *X = 1, Y = 0*
- *X = 1, Y = 1*
- *X = 2, Y = 1*

20.7 atleast_nvalues

The *atleast_nvalues* constraint involves:

- an array of integer variables *VARS*,
- an integer variable *NVALUES* and
- a boolean *AC*.

Let *N* be the number of distinct values assigned to the variables of the *VARS* collection. The constraint enforces the condition $N \geq NVALUES$ to hold. The boolean *AC* set to true enforces arc-consistency.

See also: atleast_nvalues in the Global Constraint Catalog.

Implementation based on: [Regin95].

API:

```
Constraint atleast_nvalues(IntVar[] VARS, IntVar NVALUES, boolean AC)
```

Example

```
        Solver solver = new Solver();
        IntVar[] XS = VF.enumeratedArray("XS", 4, 0, 2, solver);
        IntVar N = VF.enumerated("N", 2, 3, solver);
        solver.post(ICF.atleast_nvalues(XS, N, true));
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *XS[0] = 0 XS[1] = 0 XS[2] = 0 XS[3] = 1 N = 2*
- *XS[0] = 0 XS[1] = 1 XS[2] = 0 XS[3] = 1 N = 2*
- *XS[0] = 0 XS[1] = 1 XS[2] = 2 XS[3] = 1 N = 2*
- *XS[0] = 2 XS[1] = 0 XS[2] = 2 XS[3] = 1 N = 3*
- *XS[0] = 2 XS[1] = 2 XS[2] = 1 XS[3] = 0 N = 3*

20.8 atmost_nvalues

The *atmost_nvalues* constraint involves:

- an array of integer variables *VARS*,
- an integer variable *NVALUES* and
- a boolean *STRONG*.

Let N be the number of distinct values assigned to the variables of the *VARS* collection. The constraint enforces the condition $N \leq NVALUES$ to hold.

If the boolean *STRONG* is set to true, then the filtering algorithm of [FLapegue14] is added. It automatically detects disequalities and *alldifferent* constraints. This propagator is more powerful but more time consuming as well. this is presumably worthwhile when *NVALUES* must be minimized

See also: atmost_nvalues in the Global Constraint Catalog.

Implementation based on: [FLapegue14].

API:

```
Constraint atmost_nvalues(IntVar[] VARS, IntVar NVALUES, boolean GREEDY)
```

Example

```
1        Solver solver = new Solver();
2        IntVar[] XS = VF.enumeratedArray("XS", 4, 0, 2, solver);
3        IntVar N = VF.enumerated("N", 1, 3, solver);
4        solver.post(ICF.atmost_nvalues(XS, N, false));
5        solver.findAllSolutions();
```

Some solutions of the problem are :

- *XS[0] = 0, XS[1] = 0, XS[2] = 0, XS[3] = 0, N = 1*
- *XS[0] = 0, XS[1] = 0, XS[2] = 0, XS[3] = 0, N = 2*
- *XS[0] = 0, XS[1] = 0, XS[2] = 0, XS[3] = 0, N = 3*
- *XS[0] = 0, XS[1] = 0, XS[2] = 0, XS[3] = 1, N = 2*
- *XS[0] = 0, XS[1] = 1, XS[2] = 1, XS[3] = 0, N = 2*
- *XS[0] = 2, XS[1] = 2, XS[2] = 1, XS[3] = 0, N = 3*

20.9 bin_packing

The *bin_packing* constraint involves:

- an array of integer variables *ITEM_BIN*,
- an array of integers *ITEM_SIZE*,
- an array of integer variables *BIN_LOAD* and
- an integer *OFFSET*.

It holds the Bin Packing Problem rules: a set of items with various SIZES to pack into bins with respect to the capacity of each bin.

- *ITEM_BIN* represents the bin of each item, that is, *ITEM_BIN[i] = j* states that the i th ITEM is put in the j th bin.
- *ITEM_SIZE* represents the size of each item.
- *BIN_LOAD* represents the load of each bin, that is, the sum of size of the items in it.

This constraint is not a built-in constraint and is based on various propagators.

See also: bin_packing in the Global Constraint Catalog.

API:

```
Constraint[] bin_packing(IntVar[] ITEM_BIN, int[] ITEM_SIZE, IntVar[] BIN_LOAD, int OFFSET)
```

Example

```
1         Solver solver = new Solver();
2         IntVar[] IBIN = VF.enumeratedArray("IBIN", 5, 1, 3, solver);
3         int[] sizes = new int[]{2, 3, 1, 4, 2};
4         IntVar[] BLOADS = VF.enumeratedArray("BLOADS", 3, 0, 5, solver);
5         solver.post(ICF.bin_packing(IBIN, sizes, BLOADS, 1));
6         solver.findAllSolutions();
```

Some solutions of the problem are :

- *IBIN[0] = 1, IBIN[1] = 1, IBIN[2] = 2, IBIN[3] = 2, IBIN[4] = 3, BLOADS[0] = 5, BLOADS[1] = 5, BLOADS[2] = 2*
- *IBIN[0] = 1, IBIN[1] = 3, IBIN[2] = 1, IBIN[3] = 2, IBIN[4] = 1, BLOADS[0] = 5, BLOADS[1] = 4, BLOADS[2] = 3*
- *IBIN[0] = 2, IBIN[1] = 3, IBIN[2] = 1, IBIN[3] = 1, IBIN[4] = 3, BLOADS[0] = 5, BLOADS[1] = 2, BLOADS[2] = 5*

20.10 bit_channeling

The *bit_channeling* constraint involves:

- an array of boolean variables *BVARS* and
- an integer variable *VAR*.

It ensures that: *VAR* = $2^0 \times$ *BITS[0]* $2^1 \times$ *BITS[1]* + ... +:math:*2^{n} times BITS[n]. BIT[0] is related to the first bit of 'VAR* (2^0), *BIT[1] is related to the second bit of 'VAR* (2^1), etc. The upper bound of *VAR* is given by $2^{|BITS|+1}$.

API:

```
Constraint bit_channeling(BoolVar[] BITS, IntVar VAR)
```

Example

```
        Solver solver = new Solver();
        BoolVar[] BVARS = VF.boolArray("BVARS", 4, solver);
        IntVar VAR = VF.enumerated("VAR", 0, 15, solver);
        solver.post(ICF.bit_channeling(BVARS, VAR));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *VAR = 0, BVARS[0] = 0, BVARS[1] = 0, BVARS[2] = 0, BVARS[3] = 0*
- *VAR = 1, BVARS[0] = 1, BVARS[1] = 0, BVARS[2] = 0, BVARS[3] = 0*
- *VAR = 2, BVARS[0] = 0, BVARS[1] = 1, BVARS[2] = 0, BVARS[3] = 0*
- *VAR = 11, BVARS[0] = 1, BVARS[1] = 1, BVARS[2] = 0, BVARS[3] = 1*
- *VAR = 15, BVARS[0] = 1, BVARS[1] = 1, BVARS[2] = 1, BVARS[3] = 1*

20.11 boolean_channeling

The *boolean_channeling* constraint involves:

- an array of boolean variables *BVARS*,
- an integer variable *VAR* and
- an integer *OFFSET*.

It ensures that: *VAR = i* $\Leftrightarrow$ *BVARS* [*i-OFFSET*] = *1*. The *OFFSET* is typically set to 0.

API:

```
Constraint boolean_channeling(BoolVar[] BVARS, IntVar VAR, int OFFSET)
```

Example

```
        Solver solver = new Solver();
        BoolVar[] BVARS = VF.boolArray("BVARS", 5, solver);
        IntVar VAR = VF.enumerated("VAR", 1, 5, solver);
        solver.post(ICF.boolean_channeling(BVARS, VAR, 1));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *VAR = 1, BVARS[0] = 1, BVARS[1] = 0, BVARS[2] = 0, BVARS[3] = 0, BVARS[4] = 0*
- *VAR = 2, BVARS[0] = 0, BVARS[1] = 1, BVARS[2] = 0, BVARS[3] = 0, BVARS[4] = 0*
- *VAR = 3, BVARS[0] = 0, BVARS[1] = 0, BVARS[2] = 1, BVARS[3] = 0, BVARS[4] = 0*
- *VAR = 4, BVARS[0] = 0, BVARS[1] = 0, BVARS[2] = 0, BVARS[3] = 1, BVARS[4] = 0*
- *VAR = 5, BVARS[0] = 0, BVARS[1] = 0, BVARS[2] = 0, BVARS[3] = 0, BVARS[4] = 1*

20.12 clause_channeling

The *clause_channeling* constraint involves:

- an integer variable *VAR* and
- two arrays of boolean variables *EVARS* and *LVARS*.

It ensures that: $VAR = i \Leftrightarrow EVARS[i - OFFSET] = 1$ **and** $VAR \leq i \Leftrightarrow LVARS[i - OFFSET] = 1$ where OFFSET is the initial lower bound of VAR.

API:

```
Constraint clause_channeling(IntVar VAR, BoolVar[] EVARS, BoolVar[] LVARS)
```

Example

```
1    public void clause_channeling() {
2        Solver solver = new Solver();
3        IntVar iv = VF.enumerated("iv", 1, 3, solver);
4        BoolVar[] eqs = VF.boolArray("eq", 3, solver);
5        BoolVar[] lqs = VF.boolArray("lq", 3, solver);
6        Chatterbox.showSolutions(solver);
```

The solutions of the problem are :

- *VAR = 1, EVARS[0] = 1, EVARS[1] = 0, EVARS[2] = 0, LVARS[0] = 1, LVARS[1] = 1, LVARS[2] = 1*
- *VAR = 2, EVARS[0] = 0, EVARS[1] = 1, EVARS[2] = 0, LVARS[0] = 0, LVARS[1] = 1, LVARS[2] = 1*
- *VAR = 3, EVARS[0] = 0, EVARS[1] = 0, EVARS[2] = 1, LVARS[0] = 0, LVARS[1] = 0, LVARS[2] = 1*

20.13 circuit

The *circuit* constraint involves:

- an array of integer variables *VARS*,
- an integer *OFFSET* and
- a configuration *CONF*.

It ensures that the elements of *VARS* define a covering circuit where *VARS* [i] = *OFFSET* + *j* means that *j* is the successor of *i*.

The filtering algorithms are the subtour elimination of [CL97] (constant-time per propagation) and the *alldifferent* GAC filtering of [Regin94]. In addition, depending on *CONF*, the dominator filtering of the tree (GAC) constraint [FL11] and the strongly connected components filtering of the path constraint [CB04][FL12] may be added through a dynamical circuit/path transformation.

The *CONF* is a defined by an `enum`:

- `CircuitConf.LIGHT`: no circuit/path transformation
- `CircuitConf.FIRST`: circuit/path transformation by duplicating the first node
- `CircuitConf.RD`: circuit/path transformation by duplicating a random node
- `CircuitConf.ALL`: circuit/path transformation by duplicating every node

This implementation is detailed in [Fag14]

See also: circuit in the Global Constraint Catalog.

Implementation based on: [Regin94][CL97][CB04][FL12][FL11][Fag14].

API:

```
Constraint circuit(IntVar[] VARS, int OFFSET, CircuitConf CONF)
Constraint circuit(IntVar[] VARS, int OFFSET) // with CircuitConf.RD
```

Example

```
1        Solver solver = new Solver();
2        IntVar[] NODES = VF.enumeratedArray("NODES", 5, 0, 4, solver);
3        solver.post(ICF.circuit(NODES, 0, CircuitConf.LIGHT));
4        solver.findAllSolutions();
```

Some solutions of the problem are :

- *NODES[0] = 1, NODES[1] = 2, NODES[2] = 3, NODES[3] = 4, NODES[4] = 0*
- *NODES[0] = 3, NODES[1] = 4, NODES[2] = 0, NODES[3] = 1, NODES[4] = 2*
- *NODES[0] = 4, NODES[1] = 2, NODES[2] = 3, NODES[3] = 0, NODES[4] = 1*
- *NODES[0] = 4, NODES[1] = 3, NODES[2] = 1, NODES[3] = 0, NODES[4] = 2*

20.14 cost_regular

The *cost_regular* constraint involves:

- an array of integer variables *VARS*,
- an integer variable *COST* and
- a cost automaton *CAUTOMATON*.

It ensures that the assignment of a sequence of variables *VARS* is recognized by *CAUTOMATON*, a deterministic finite automaton, and that the sum of the costs associated to each assignment is bounded by the cost variable. This version allows to specify different costs according to the automaton state at which the assignment occurs (i.e. the transition starts).

The *CAUOTMATON* can be defined using the *org.chocosolver.solver.constraints.nary.automata.FA.CostAutomaton* either:

- by creating a `CostAutomaton`: once created, states should be added, then initial and final states are defined and finally, transitions are declared.
- or by first creating a `FiniteAutomaton` and then creating a matrix of costs and finally calling one of the following API from `CostAutomaton`:
  - `ICostAutomaton makeSingleResource(IAutomaton pi, int[][][] costs, int inf, int sup)`
  - `ICostAutomaton makeSingleResource(IAutomaton pi, int[][] costs, int inf, int sup)`

The other API of `CostAutomaton` (`makeMultiResources(...)`) are dedicated to the *multi-cost_regular* constraint.

Implementation based on: [DPR06].

API:

```
Constraint cost_regular(IntVar[] VARS, IntVar COST, ICostAutomaton CAUTOMATON)
```

Example

```
        Solver solver = new Solver();
        IntVar[] VARS = VF.enumeratedArray("VARS", 5, 0, 2, solver);
        IntVar COST = VF.enumerated("COST", 0, 10, solver);
        FiniteAutomaton fauto = new FiniteAutomaton();
        int start = fauto.addState();
        int end = fauto.addState();
        fauto.setInitialState(start);
        fauto.setFinal(start, end);

        fauto.addTransition(start, start, 0, 1);
        fauto.addTransition(start, end, 2);

        fauto.addTransition(end, end, 1);
        fauto.addTransition(end, start, 0, 2);

        int[][] costs = new int[5][3];
        costs[0] = new int[]{1, 2, 3};
        costs[1] = new int[]{2, 3, 1};
        costs[2] = new int[]{3, 1, 2};
        costs[3] = new int[]{3, 2, 1};
        costs[4] = new int[]{2, 1, 3};

        solver.post(ICF.cost_regular(VARS, COST, CostAutomaton.makeSingleResource(fauto, costs,
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *VARS[0] = 0, VARS[1] = 0, VARS[2] = 0, VARS[3] = 0, VARS[4] = 1, COST = 10*
- *VARS[0] = 0, VARS[1] = 0, VARS[2] = 0, VARS[3] = 1, VARS[4] = 1, COST = 9*
- *VARS[0] = 0, VARS[1] = 0, VARS[2] = 1, VARS[3] = 2, VARS[4] = 1, COST = 6*
- *VARS[0] = 1, VARS[1] = 2, VARS[2] = 1, VARS[3] = 0, VARS[4] = 1, COST = 8*

20.15 count

The *count* constraint involves:

- an integer *VALUE*,
- an array of integer variables *VARS* and
- an integer variable *LIMIT*.

The constraint holds that *LIMIT* is equal to the number of variables from *VARS* assigned to the value *VALUE*. An alternate signature enables *VALUE* to be an integer variable.

See also: count in the Global Constraint Catalog.

API:

```
Constraint count(int VALUE, IntVar[] VARS, IntVar LIMIT)
Constraint count(IntVar VALUE, IntVar[] VARS, IntVar LIMIT)
```

Example

```
1         Solver solver = new Solver();
2         IntVar[] VS = VF.enumeratedArray("VS", 4, 0, 3, solver);
3         IntVar VA = VF.enumerated("VA", new int[]{1, 3}, solver);
4         IntVar CO = VF.enumerated("CO", new int[]{0, 2, 4}, solver);
5         solver.post(ICF.count(VA, VS, CO));
6         solver.findAllSolutions();
```

Some solutions of the problem are :

- *VS[0] = 0, VS[1] = 0, VS[2] = 0, VS[3] = 0, VA = 1, CO = 0*
- *VS[0] = 0, VS[1] = 1, VS[2] = 1, VS[3] = 0, VA = 1, CO = 2*
- *VS[0] = 0, VS[1] = 2, VS[2] = 2, VS[3] = 1, VA = 3, CO = 0*
- *VS[0] = 3, VS[1] = 3, VS[2] = 3, VS[3] = 3, VA = 3, CO = 4*

20.16 cumulative

The *cumulative* constraints involves:

- an array of task object *TASKS*,
- an array of integer variable *HEIGHTS*,
- an integer variable *CAPACITY* and
- a boolean *INCREMENTAL* (graph-based self-decomposition of [FLP14]).

It ensures that at each point of the time the cumulative height of the set of tasks that overlap that point does not exceed the given capacity.

See also: cumulative in the Global Constraint Catalog.

Implementation based on: [FLP14].

API:

```
Constraint cumulative(Task[] TASKS, IntVar[] HEIGHTS, IntVar CAPACITY)
Constraint cumulative(Task[] TASKS, IntVar[] HEIGHTS, IntVar CAPACITY, boolean INCREMENTAL)
```

The first API relies on the second, and set *INCREMENTAL* to `TASKS.length > 500`.

Example 1

```
1         Solver solver = new Solver();
2         Task[] TS = new Task[5];
3         IntVar[] HE = new IntVar[5];
4         for (int i = 0; i < TS.length; i++) {
```

```
            IntVar S = VF.bounded("S_" + i, 0, 4, solver);
            TS[i] = VF.task(
                    S,
                    VF.fixed("D_" + i, i + 1, solver),
                    VF.offset(S, i + 1)
            );
            HE[i] = VF.bounded("HE_" + i, i - 1, i + 1, solver);
        }
        IntVar CA = VF.enumerated("CA", 1, 3, solver);
        solver.post(ICF.cumulative(TS, HE, CA, true));
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *S_0 = 0, HE_0 = 0, S_1 = 0, HE_1 = 0, S_2 = 0, HE_2 = 1, S_3 = 0, HE_3 = 2 S_4 = 4, HE_4 = 3, CA = 3*
- *S_0 = 4, HE_0 = 0, S_1 = 4, HE_1 = 0, S_2 = 1, HE_2 = 1, S_3 = 0, HE_3 = 2 S_4 = 4, HE_4 = 3, CA = 3*
- *S_0 = 0, HE_0 = 1, S_1 = 0, HE_1 = 0, S_2 = 1, HE_2 = 1, S_3 = 0, HE_3 = 2 S_4 = 4, HE_4 = 3, CA = 3*

20.17 diffn

The *diffn* constraint involves:

- four arrays of integer variables *X*, *Y*, *WIDTH* and *HEIGHT* and
- a boolean *USE_CUMUL*.

It ensures that each rectangle *i* defined by its coordinates (*X[i]*, *Y[i]*) and its dimensions (*WIDTH[i]*, *HEIGHT[i]*) does not overlap each other. The option *USE_CUMUL*, recommended, indicates whether or not redundant *cumulative* constraints should be added on each dimension.

See also: diffn in the Global Constraint Catalog.

Implementation based on: [FLP14].

API:

```
Constraint[] diffn(IntVar[] X, IntVar[] Y, IntVar[] WIDTH, IntVar[] HEIGHT, boolean USE_CUMUL)
```

Example 1

```
        Solver solver = new Solver();
        IntVar[] X = VF.boundedArray("X", 4, 0, 1, solver);
        IntVar[] Y = VF.boundedArray("Y", 4, 0, 2, solver);
        IntVar[] D = new IntVar[4];
        IntVar[] W = new IntVar[4];
        for (int i = 0; i < 4; i++) {
            D[i] = VF.fixed("D_" + i, 1, solver);
            W[i] = VF.fixed("W_" + i, i + 1, solver);
        }
        solver.post(ICF.diffn(X, Y, D, W, true));
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *X[0] = 0 X[1] = 1, X[2] = 0, X[3] = 1, Y[0] = 0, Y[1] = 0, Y[2] = 1, Y[3]*
- *X[0] = 1 X[1] = 0, X[2] = 1, X[3] = 0, Y[0] = 0, Y[1] = 0, Y[2] = 2, Y[3]*
- *X[0] = 0 X[1] = 1, X[2] = 0, X[3] = 1, Y[0] = 1, Y[1] = 0, Y[2] = 2, Y[3]*

20.18 distance

The `distance` constraint involves either:

- two variables *VAR1* and *VAR2*, an operator *OP* and a constant *CSTE*. It ensures that | *VAR1* - *VAR2* | *OP CSTE*, where *OP* must be chosen in `{"=", "!=", ">","<"}` .
- or three variables *VAR1*, *VAR2* and *VAR3* and an operator *OP*. It ensures that | *VAR1* - *VAR2* | *OP VAR3*, where *OP* must be chosen in `{"=",">","<"}` .

See also: distance in the Global Constraint Catalog.

API:

```
Constraint distance(IntVar VAR1, IntVar VAR2, String OP, int CSTE)
Constraint distance(IntVar VAR1, IntVar VAR2, String OP, IntVar VAR3)
```

Example 1

```
1        Solver solver = new Solver();
2        IntVar X = VF.enumerated("X", 0, 2, solver);
3        IntVar Y = VF.enumerated("X", -3, 1, solver);
4        solver.post(ICF.distance(X, Y, "=", 1));
5        solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 0, Y = -1*
- *X = 0, Y = 1*
- *X = 1, Y = 0*
- *X = 2, Y = 1*

Example 2

```
1        Solver solver = new Solver();
2        IntVar X = VF.enumerated("X", 1, 3, solver);
3        IntVar Y = VF.enumerated("Y", -1, 1, solver);
4        IntVar Z = VF.enumerated("Z", 2, 3, solver);
5        solver.post(ICF.distance(X, Y, "<", Z));
6        solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 1, Y = 0, Z = 2*
- *X = 1, Y = 1, Z = 2*
- *X = 2, Y = 1, Z = 2*

- *X = 1, Y = -1, Z = 3*
- *X = 1, Y = 0, Z = 3*
- *X = 1, Y = 1, Z = 3*
- *X = 2, Y = 0, Z = 3*
- *X = 2, Y = 1, Z = 3*
- *X = 3, Y = 1, Z = 3*

20.19 element

The *element* constraint involves either:

- two variables *VALUE* and *INDEX*, an array of values *TABLE*, an offset *OFFSET* and an ordering property *SORT*. *SORT* must be chosen among:
  - `"none"`: if values in *TABLE* are not sorted,
  - `"asc"`: if values in *TABLE* are sorted in increasing order,
  - `"desc"`: if values in *TABLE* are sorted in decreasing order,
  - `"detect"`: let the constraint detects the ordering of values in *TABLE*, if any (default value).
- or an integer variable *VALUE*, an array of integer variables *TABLE*, an integer variable *INDEX* and an integer *OFFSET*.

The *element* constraint ensures that *VALUE = TABLE* [*INDEX - OFFSET*]. *OFFSET* matches *INDEX.LB* and *TABLE[0]* (0 by default).

See also: element in the Global Constraint Catalog.

API:

```
Constraint element(IntVar VALUE, int[] TABLE, IntVar INDEX)
Constraint element(IntVar VALUE, int[] TABLE, IntVar INDEX, int OFFSET, String SORT)
Constraint element(IntVar VALUE, IntVar[] TABLE, IntVar INDEX, int OFFSET)
```

Example

```
1        Solver solver = new Solver();
2        IntVar V = VF.enumerated("V", -2, 2, solver);
3        IntVar I = VF.enumerated("I", 0, 5, solver);
4        solver.post(ICF.element(V, new int[]{2, -2, 1, -1, 0}, I, 0, "none"));
5        solver.findAllSolutions();
```

The solutions of the problem are :

- *V = -2, I = 1*
- *V = -1, I = 3*
- *V = 0, I = 4*
- *V = 1, I = 2*
- *V = 2, I = 0*

20.20 eucl_div

The *eucl_div* constraints involves three variables *DIVIDEND*, *DIVISOR* and *RESULT*. It ensures that *DIVIDEND* / *DIVISOR* = *RESULT*, rounding towards 0.

The API is :

```
Constraint eucl_div(IntVar DIVIDEND, IntVar DIVISOR, IntVar RESULT)
```

Example

```
        Solver solver = new Solver();
        IntVar X = VF.enumerated("X", 1, 3, solver);
        IntVar Y = VF.enumerated("Y", -1, 1, solver);
        IntVar Z = VF.enumerated("Z", 2, 3, solver);
        solver.post(ICF.eucl_div(X, Y, Z));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 2, Y = 1, Z = 2*
- *X = 3, Y = 1, Z = 3*

20.21 FALSE

The *FALSE* constraint is always unsatisfied. It should only be used with `LogicalFactory`.

20.22 global_cardinality

The *global_cardinality* constraint involves:

- an array of integer variables *VARS*,
- an array of integer *VALUES*,
- an array of integer variables *OCCURRENCES* and
- a boolean *CLOSED*.

It ensures that each value *VALUES[i]* is taken by exactly *OCCURRENCES[i]* variables in *VARS*. The boolean *CLOSED* set to *true* restricts the domain of *VARS* to the values defined in *VALUES*.

The underlying propagator does not ensure any well-defined level of consistency.

See also: global_cardinality in the Global Constraint Catalog.

API:

```
Constraint global_cardinality(IntVar[] VARS, int[] VALUES, IntVar[] OCCURRENCES, boolean CLOSED)
```

Example

```
        Solver solver = new Solver();
        IntVar[] VS = VF.boundedArray("VS", 4, 0, 4, solver);
        int[] values = new int[]{-1, 1, 2};
        IntVar[] OCC = VF.boundedArray("OCC", 3, 0, 2, solver);
        solver.post(ICF.global_cardinality(VS, values, OCC, true));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *VS[0] = 1, VS[1] = 1, VS[2] = 2, VS[3] = 2, OCC[0] = 0, OCC[1] = 2, OCC[2] = 2*
- *VS[0] = 1, VS[1] = 2, VS[2] = 1, VS[3] = 2, OCC[0] = 0, OCC[1] = 2, OCC[2] = 2*
- *VS[0] = 1, VS[1] = 2, VS[2] = 2, VS[3] = 1, OCC[0] = 0, OCC[1] = 2, OCC[2] = 2*
- *VS[0] = 2, VS[1] = 1, VS[2] = 1, VS[3] = 2, OCC[0] = 0, OCC[1] = 2, OCC[2] = 2*
- *VS[0] = 2, VS[1] = 1, VS[2] = 2, VS[3] = 1, OCC[0] = 0, OCC[1] = 2, OCC[2] = 2*
- *VS[0] = 2, VS[1] = 2, VS[2] = 1, VS[3] = 1, OCC[0] = 0, OCC[1] = 2, OCC[2] = 2*

20.23 inverse_channeling

The *inverse_channeling* constraint involves:

- two arrays of integer variables *VARS1* and *VARS2* and
- two integers *OFFSET1* and *OFFSET2*.

It ensures that *VARS1[i - OFFSET2] = j* ⇔ *VARS2[j - OFFSET1] = i*. It performs AC if the domains are enumerated. Otherwise, BC is not guaranteed. It also automatically imposes one *alldifferent* constraints on each array of variables.

API:

```
Constraint inverse_channeling(IntVar[] VARS1, IntVar[] VARS2, int OFFSET1, int OFFSET2)
```

Example

```
        Solver solver = new Solver();
        IntVar[] X = VF.enumeratedArray("X", 3, 0, 3, solver);
        IntVar[] Y = VF.enumeratedArray("Y", 3, 1, 4, solver);
        solver.post(ICF.inverse_channeling(X, Y, 0, 1));
        solver.findAllSolutions();
```

The solutions of the problems are:

- *X[0] = 0, X[1] = 1, X[2] = 2, Y[0] = 1, Y[1] = 2, Y[2] = 3*
- *X[0] = 0, X[1] = 2, X[2] = 1, Y[0] = 1, Y[1] = 3, Y[2] = 2*
- *X[0] = 1, X[1] = 0, X[2] = 2, Y[0] = 2, Y[1] = 1, Y[2] = 3*
- *X[0] = 1, X[1] = 2, X[2] = 0, Y[0] = 3, Y[1] = 1, Y[2] = 2*
- *X[0] = 2, X[1] = 0, X[2] = 1, Y[0] = 2, Y[1] = 3, Y[2] = 1*
- *X[0] = 2, X[1] = 1, X[2] = 0, Y[0] = 3, Y[1] = 2, Y[2] = 1*

20.24 int_value_precede_chain

The *int_value_precede_chain* constraint involves an array of integer variables *X* and

- either two integers *S* and *T*
- or an array of distinct integers.

It ensures that if there exists j such that X[j] = T, then, there must exist i < j such that X[i] = S. Or it ensures that, for each pair of V[k] and V[l] of values in V, such that k < l, if there exists j such that X[j] = V[l], then, there must exist i < j such that X[i] = V[k].

See also: int_value_precede in the Global Constraint Catalog.

Implementation based on: [LL04].

** API**:

```
Constraint int_value_precede_chain(IntVar[] X, int S, int T)
```

Example

```
1         Solver solver = new Solver();
2         IntVar[] X = VF.enumeratedArray("X", 3, 1, 3, solver);
3         solver.post(ICF.int_value_precede_chain(X, new int[]{2,3,1}));
4         solver.findAllSolutions();
```

The solutions of the problems are:

- *X[0] = 2 X[1] = 2 X[2] = 2*
- *X[0] = 2 X[1] = 2 X[2] = 3*
- *X[0] = 2 X[1] = 3 X[2] = 1*
- *X[0] = 2 X[1] = 3 X[2] = 2*
- *X[0] = 2 X[1] = 3 X[2] = 3*

20.25 keysorting

The *keysorting* constraint involves three matrices of integer variables *VARS* and *SORTEDVARS*, an array of integer variables *PERMVARS* and an integer *K*. It ensures that the variables of *SORTEDVARS* correspond to the variables of *VARS* according to a permutation. Moreover, the variable of *SORTEDVARS* are sorted in increasing order wrt to K-tuple and PERMVARS store the permutations.

API:

```
Constraint keysorting(IntVar[][] VARS, IntVar[] PERMVARS, IntVar[][] SORTEDVARS, int K)
```

Example

```
1         Solver solver = new Solver();
2         IntVar[] X = VF.enumeratedArray("X", 3, 1, 3, solver);
3         solver.post(ICF.int_value_precede_chain(X, 1, 2));
```

```
        Chatterbox.showSolutions(solver);
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *X[0][0] = 2 X[0][1] = 1 X[0][2] = 1 X[1][0] = 1 X[1][1] = 1 X[1][2] = 1*

Y[0][0] = 1 Y[0][1] = 1 Y[0][2] = 1 Y[1][0] = 2 Y[1][1] = 1 Y[1][2] = 1 P[0] = 2 P[1] = 1 P[2] = 0

- *X[0][0] = 2 X[0][1] = 1 X[0][2] = 1 X[1][0] = 1 X[1][1] = 3 X[1][2] = 2*

Y[0][0] = 1 Y[0][1] = 3 Y[0][2] = 2 Y[1][0] = 2 Y[1][1] = 1 Y[1][2] = 1 P[0] = 2 P[1] = 1 P[2] = 2

- *X[0][0] = 2 X[0][1] = 1 X[0][2] = 3 X[1][0] = 1 X[1][1] = 1 X[1][2] = 2*

Y[0][0] = 1 Y[0][1] = 1 Y[0][2] = 2 Y[1][0] = 2 Y[1][1] = 1 Y[1][2] = 3 P[0] = 2 P[1] = 1 P[2] = 1

20.26 knapsack

The *knapsack* constraint involves: - an array of integer variables *OCCURRENCES*, - an integer variable *TOTAL_WEIGHT*, - an integer variable *TOTAL_ENERGY*, - an array of integers *WEIGHT* and - an an array of integers *ENERGY*.

It formulates the Knapsack Problem: to determine the count of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

- $\sum$ *OCCURRENCES[i]* $\times$ *WEIGHT[i]* $\leq$ *TOTAL_WEIGHT* and
- $\sum$ *OCCURRENCES[i]* $\times$ *ENERGY[i]* = *TOTAL_ENERGY*.

API:

```
Constraint knapsack(IntVar[] OCCURRENCES, IntVar TOTAL_WEIGHT, IntVar TOTAL_ENERGY,
                                    int[] WEIGHT, int[] ENERGY)
```

Example

```
        Solver solver = new Solver();
        IntVar[] IT = new IntVar[3]; // 3 items
        IT[0] = VF.bounded("IT_0", 0, 3, solver);
        IT[1] = VF.bounded("IT_1", 0, 2, solver);
        IT[2] = VF.bounded("IT_2", 0, 1, solver);
        IntVar WE = VF.bounded("WE", 0, 8, solver);
        IntVar EN = VF.bounded("EN", 0, 6, solver);
        int[] weights = new int[]{1, 3, 4};
        int[] energies = new int[]{1, 4, 6};
        solver.post(ICF.knapsack(IT, WE, EN, weights, energies));
        solver.findAllSolutions();
```

Some solutions of the problems are:

- *IT_0 = 0, IT_1 = 0, IT_2 = 0, WE = 0, EN = 0*
- *IT_0 = 3, IT_1 = 0, IT_2 = 0, WE = 3, EN = 3*
- *IT_0 = 1, IT_1 = 1, IT_2 = 0, WE = 4, EN = 5*
- *IT_0 = 2, IT_1 = 1, IT_2 = 0, WE = 5, EN = 6*

20.27 lex_chain_less

The *lex_chain_less* constraint involves a matrix of integer variables *VARS*. It ensures that, for each pair of consecutive arrays *VARS[i]* and *VARS[i+1]*, *VARS[i]* is lexicographically strictly less than *VARS[i+1]*.

See also: lex_chain_less in the Global Constraint Catalog.

Implementation based on: [CB02].

API:

```
Constraint lex_chain_less(IntVar[]... VARS)
```

Example

```
        Solver solver = new Solver();
        IntVar[] X = VF.enumeratedArray("X", 3, -1, 1, solver);
        IntVar[] Y = VF.enumeratedArray("Y", 3, 1, 2, solver);
        IntVar[] Z = VF.enumeratedArray("Z", 3, 0, 2, solver);
        solver.post(ICF.lex_chain_less(X, Y, Z));
        solver.findAllSolutions();
```

Some solutions of the problems are:

- *X[0] = -1, X[1] = -1, X[2] = -1, Y[0] = 1, Y[1] = 1, Y[2] = 1, Z[0] = 1, Z[1] = 1, Z[2] = 2*
- *X[0] = 0, X[1] = 1, X[2] = -1, Y[0] = 1, Y[1] = 1, Y[2] = 1, Z[0] = 1, Z[1] = 2, Z[2] = 0*
- *X[0] = 1, X[1] = 0, X[2] = 1, Y[0] = 1, Y[1] = 1, Y[2] = 1, Z[0] = 1, Z[1] = 2, Z[2] = 0*
- *X[0] = -1, X[1] = 1, X[2] = 1, Y[0] = 1, Y[1] = 1, Y[2] = 1, Z[0] = 2, Z[1] = 2, Z[2] = 1*

20.28 lex_chain_less_eq

The *lex_chain_less_eq* constraint involves a matrix of integer variables *VARS*. It ensures that, for each pair of consecutive arrays *VARS[i]* and *VARS[i+1]*, *VARS[i]* is lexicographically strictly less or equal than *VARS[i+1]*.

See also: lex_chain_less_eq in the Global Constraint Catalog.

Implementation based on: [CB02].

API:

```
Constraint lex_chain_less_eq(IntVar[]... VARS)
```

Example

```
        Solver solver = new Solver();
        IntVar[] X = VF.enumeratedArray("X", 3, -1, 1, solver);
        IntVar[] Y = VF.enumeratedArray("Y", 3, 1, 2, solver);
        IntVar[] Z = VF.enumeratedArray("Z", 3, 0, 2, solver);
```

```
5         solver.post(ICF.lex_chain_less_eq(X, Y, Z));
6         solver.findAllSolutions();
```

Some solutions of the problems are:

- *X[0] = -1, X[1] = -1, X[2] = -1, Y[0] = 1, Y[1] = 1, Y[2] = 1, Z[0] = 1, Z[1] = 1, Z[2] = 1*
- *X[0] = -1, X[1] = 1, X[2] = 1, Y[0] = 1, Y[1] = 1, Y[2] = 1, Z[0] = 1, Z[1] = 1, Z[2] = 1*
- *X[0] = 0, X[1] = 1, X[2] = -1, Y[0] = 1, Y[1] = 1, Y[2] = 1, Z[0] = 2, Z[1] = 1, Z[2] = 2*
- *X[0] = -1, X[1] = -1, X[2] = 0, Y[0] = 1, Y[1] = 1, Y[2] = 2, Z[0] = 2, Z[1] = 2, Z[2] = 2*

20.29 lex_less

The *lex_less* constraint involves two arrays of integer variables *VARS1* and *VARS2*. It ensures that *VARS1* is lexicographically strictly less than *VARS2*.

See also: lex_less in the Global Constraint Catalog.

Implementation based on: [FHK+02].

API:

```
Constraint lex_less(IntVar[] VARS1, IntVar[] VARS2)
```

Example

```
1         Solver solver = new Solver();
2         IntVar[] X = VF.enumeratedArray("X", 3, -1, 1, solver);
3         IntVar[] Y = VF.enumeratedArray("Y", 3, 1, 2, solver);
4         solver.post(ICF.lex_less(X, Y));
5         solver.findAllSolutions();
```

Some solutions of the problems are:

- *X[0] = -1, X[1] = -1, X[2] = -1, Y[0] = 1, Y[1] = 1, Y[2] = 1*
- *X[0] = -1, X[1] = 0, X[2] = 0, Y[0] = 1, Y[1] = 2, Y[2] = 1*
- *X[0] = -1, X[1] = 0, X[2] = -1, Y[0] = 2, Y[1] = 1, Y[2] = 1*
- *X[0] = -1, X[1] = -1, X[2] = 0, Y[0] = 2, Y[1] = 2, Y[2] = 2*

20.30 lex_less_eq

The *lex_less_eq* constraint involves two arrays of integer variables *VARS1* and *VARS2*. It ensures that *VARS1* is lexicographically strictly less or equal than *VARS2*.

See also: lex_less_eq in the Global Constraint Catalog.

Implementation based on: [FHK+02].

API:

```
Constraint lex_less_eq(IntVar[] VARS1, IntVar[] VARS2)
```

Example

```
        Solver solver = new Solver();
        IntVar[] X = VF.enumeratedArray("X", 3, -1, 1, solver);
        IntVar[] Y = VF.enumeratedArray("Y", 3, 1, 2, solver);
        solver.post(ICF.lex_less_eq(X, Y));
        solver.findAllSolutions();
```

Some solutions of the problems are:

- *X[0] = -1, X[1] = -1, X[2] = -1, Y[0] = 1, Y[1] = 1, Y[2] = 1*
- *X[0] = 1, X[1] = -1, X[2] = -1, Y[0] = 1, Y[1] = 1, Y[2] = 1*
- *X[0] = 0, X[1] = 0, X[2] = 0, Y[0] = 2, Y[1] = 1, Y[2] = 2*
- *X[0] = 1, X[1] = 1, X[2] = 1, Y[0] = 2, Y[1] = 2, Y[2] = 2*

20.31 maximum

The *maximum* constraints involves a set of integer variables and a third party integer variable, either:

- two integer variables *VAR1* and *VAR2* and an integer variable *MAX*, it ensures that *MAX*'= *maximum*(*'VAR1*, *VAR2*).
- or an array of integer variables *VARS* and an integer variable *MAX*, it ensures that *MAX* is the maximum value of the collection of domain variables *VARS*.
- or an array of boolean variables *BVARS* and a booean variable *MAX*, it ensures that *MAX* is the maximum value of the collection of boolean variables *BVARS*.

See also: maximum in the Global Constraint Catalog.

API:

```
Constraint maximum(IntVar MAX, IntVar VAR1, IntVar VAR2)
Constraint maximum(IntVar MAX, IntVar[] VARS)
Constraint maximum(BoolVar MAX, BoolVar[] VARS)
```

Example

```
        Solver solver = new Solver();
        IntVar MAX = VF.enumerated("MAX", 1, 3, solver);
        IntVar Y = VF.enumerated("Y", -1, 1, solver);
        IntVar Z = VF.enumerated("Z", 2, 3, solver);
        solver.post(ICF.maximum(MAX, Y, Z));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *MAX = 2, Y = -1, Z = 2*
- *MAX = 2, Y = 0, Z = 2*

- *MAX = 2, Y = 1, Z = 2*
- *MAX = 3, Y = -1, Z = 3*
- *MAX = 3, Y = 0, Z = 3*
- *MAX = 3, Y = 1, Z = 3*

20.32 mddc

A constraint which restricts the values a variable can be assigned to the solutions encoded with a multi-valued decision diagram.

Implementation based on: [CY08].

API:

```
Constraint mddc(IntVar[] VARS, MultivaluedDecisionDiagram MDD)
```

Example

```
1    public void mddc() {
2        Solver solver = new Solver();
3        IntVar[] vars = VF.enumeratedArray("X", 2, -2, 2, solver);
4        Tuples tuples = new Tuples();
5        tuples.add(0, -1);
6        tuples.add(1, -1);
7        tuples.add(0, 1);
8        Chatterbox.showSolutions(solver);
```

The solutions of the problem are :

- *X[0] = 0, X[1] = -1*
- *X[0] = 0, X[1] = 1*,
- *X[0] = 1, X[1] = -1*

20.33 member

A constraint which restricts the values a variable can be assigned to with respect to either:

- a given list of values, it involves a integer variable *VAR* and an array of distinct values *TABLE*. It ensures that *VAR* takes its values in *TABLE*.
- or two bounds (included), it involves a integer variable *VAR* and two integer *LB* and *UB*. It ensures that *VAR* takes its values in [*LB*, *UB*].

API:

```
Constraint member(IntVar VAR, int[] TABLE)
Constraint member(IntVar VAR, int LB, int UB)
```

Example 1

```
        Solver solver = new Solver();
        IntVar X = VF.enumerated("X", 1, 4, solver);
        solver.post(ICF.member(X, new int[]{-2, -1, 0, 1, 2}));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 1*
- *X = 2*

Example 2

```
        Solver solver = new Solver();
        IntVar X = VF.enumerated("X", 1, 4, solver);
        solver.post(ICF.member(X, 2, 5));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 2*
- *X = 3*
- *X = 4*

20.34 minimum

The *minimum* constraints involves a set of integer variables and a third party integer variable, either:

- two integer variables *VAR1* and *VAR2* and an integer variable *MIN*, it ensures that *MIN*‘= *minimum*(‘*VAR1*, *VAR2*).
- or an array of integer variables *VARS* and an integer variable *MIN*, it ensures that *MIN* is the minimum value of the collection of domain variables *VARS*.
- or an array of boolean variables *BVARS* and a booean variable *MIN*, it ensures that *MIN* is the minimum value of the collection of boolean variables *BVARS*.

See also: minimum in the Global Constraint Catalog.

API: :: Constraint minimum(IntVar MIN, IntVar VAR1, IntVar VAR2) Constraint minimum(IntVar MIN, IntVar[] VARS) Constraint minimum(BoolVar MIN, BoolVar[] VARS)

Example

```
        Solver solver = new Solver();
        IntVar MIN = VF.enumerated("MIN", 1, 3, solver);
        IntVar Y = VF.enumerated("Y", -1, 1, solver);
        IntVar Z = VF.enumerated("Z", 2, 3, solver);
        solver.post(ICF.minimum(MIN, Y, Z));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *MIN = 2, Y = -1, Z = 2*
- *MIN = 2, Y = 0, Z = 2*
- *MIN = 2, Y = 1, Z = 2*
- *MIN = 3, Y = -1, Z = 3*
- *MIN = 3, Y = 0, Z = 3*
- *MIN = 3, Y = 1, Z = 3*

20.35 mod

The *mod* constraints involves three variables *X*, *Y* and *Z*. It ensures that $X \mod Y = Z$. There is no native constraint for *mod*, so this is reformulated with the help of additional variables.

The API is :

```
Constraint mod(IntVar X, IntVar Y, IntVar Z)
```

Example

```
1         Solver solver = new Solver();
2         IntVar X = VF.enumerated("X", 2, 4, solver);
3         IntVar Y = VF.enumerated("Y", -1, 4, solver);
4         IntVar Z = VF.enumerated("Z", 1, 3, solver);
5         solver.post(ICF.mod(X, Y, Z));
6         solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 2, Y = 3, Z = 2*
- *X = 2, Y = 4, Z = 2*
- *X = 3, Y = 2, Z = 1*
- *X = 3, Y = 4, Z = 3*
- *X = 4, Y = 3, Z = 1*

20.36 multicost_regular

The *multicost_regular* constraint involves:

- an array of integer variables *VARS*,
- an array of integer variables *CVARS* and
- a cost automaton *CAUTOMATON*.

It ensures that the assignment of a sequence of variables *VARS* is recognized by *CAUTOMATON*, a deterministic finite automaton, and that the sum of the cost array associated to each assignment is bounded by the *CVARS*. This version allows to specify different costs according to the automaton state at which the assignment occurs (i.e. the transition starts).

The *CAUOTMATON* can be defined using the *org.chocosolver.solver.constraints.nary.automata.FA.CostAutomaton* either:

- by creating a `CostAutomaton`: once created, states should be added, then initial and final states are defined and finally, transitions are declared.
- or by first creating a `FiniteAutomaton` and then creating a matrix of costs and finally calling one of the following API from `CostAutomaton`:
  - `ICostAutomaton makeMultiResources(IAutomaton pi, int[][][] layer_value_resource, int[] infs, int[] sups)`
  - `ICostAutomaton makeMultiResources(IAutomaton pi, int[][][][] layer_value_resource_state, int[] infs, int[] sups)`
  - `ICostAutomaton makeMultiResources(IAutomaton auto, int[][][][] c, IntVar[] z)`
  - `ICostAutomaton makeMultiResources(IAutomaton auto, int[][][] c, IntVar[] z)`

  The other API of `CostAutomaton` (`makeSingleResource(...)`) are dedicated to the *cost_regular* constraint.

Implementation based on: [MD09].

API:

```
Constraint multicost_regular(IntVar[] VARS, IntVar[] CVARS, ICostAutomaton CAUTOMATON)
```

Example

TBD

20.37 not_member

A constraint which prevents a variable to be assigned to some values defined by either:

- a list of values, it involves a integer variable *VAR* and an array of distinct values *TABLE*. It ensures that *VAR* does not take its values in *TABLE*.
- two bounds (included), it involves a integer variable *VAR* and two integer *LB* and *UB*. It ensures that *VAR* does not take its values in [*LB*, *UB*].

The constraint

API:

```
Constraint not_member(IntVar VAR, int[] TABLE)
Constraint not_member(IntVar VAR, int LB, int UB)
```

Example 1

```
        Solver solver = new Solver();
        IntVar X = VF.enumerated("X", 1, 4, solver);
        solver.post(ICF.not_member(X, new int[]{-2, -1, 0, 1, 2}));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 3*
- *X = 4*

Example

```
        Solver solver = new Solver();
        IntVar X = VF.enumerated("X", 1, 4, solver);
        solver.post(ICF.not_member(X, 2, 5));
        solver.findAllSolutions();
```

The solution of the problem is :

- *X = 1*

20.38 nvalues

The *nvalues* constraint involves:

- an array of integer variables *VARS* and
- an integer variable *NVALUES*.

The constraint ensures that *NVALUES* is the number of distinct values assigned to the variables of the *VARS* array. This constraint is a combination of the *atleast_nvalues* and *atmost_nvalues* constraints.

This constraint is not a built-in constraint and is based on various propagators.

See also: nvalues in the Global Constraint Catalog.

Implementation based on: *atleast_nvalues* and *atmost_nvalues*.

API:

```
Constraint[] nvalues(IntVar[] VARS, IntVar NVALUES)
```

Example

```
        Solver solver = new Solver();
        IntVar[] VS = VF.enumeratedArray("VS", 4, 0, 2, solver);
        IntVar N = VF.enumerated("N", 0, 3, solver);
        solver.post(ICF.nvalues(VS, N));
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *VS[0] = 0 VS[1] = 0 VS[2] = 0 VS[3] = 0 N = 1*

- *VS[0] = 0 VS[1] = 0 VS[2] = 0 VS[3] = 1 N = 2*
- *VS[0] = 0 VS[1] = 1 VS[2] = 2 VS[3] = 2 N = 3*

20.39 path

The *path* constraint involves:

- an array of integer variables *VARS*,
- an integer variable *START*,
- an integer variable *END* and
- an integer *OFFSET*.

It ensures that the elements of *VARS* define a covering path from *START* to *END*, where *VARS[i] = OFFSET + j* means that *j* is the successor of *i*. Moreover, *VARS[END-OFFSET] = |'VARS' |+ OFFSET*.

The constraint relies on the *circuit* propagators.

See also: path in the Global Constraint Catalog.

Implementation based on: *circuit*.

API:

```
Constraint[] path(IntVar[] VARS, IntVar START, IntVar END, int OFFSET)
```

Example

```
1         Solver solver = new Solver();
2         IntVar[] VS = VF.enumeratedArray("VS", 4, 0, 4, solver);
3         IntVar S = VF.enumerated("S", 0, 3, solver);
4         IntVar E = VF.enumerated("E", 0, 3, solver);
5         solver.post(ICF.path(VS, S, E, 0));
6         solver.findAllSolutions();
```

Some solutions of the problem are :

- *VS[0] = 1, VS[1] = 2, VS[2] = 3, VS[3] = 4, S = 0, E = 3*
- *VS[0] = 1, VS[1] = 3, VS[2] = 0, VS[3] = 4, S = 2, E = 3*
- *VS[0] = 3, VS[1] = 4, VS[2] = 0, VS[3] = 1, S = 2, E = 1*
- *VS[0] = 4, VS[1] = 3, VS[2] = 1, VS[3] = 0, S = 2, E = 0*

20.40 regular

The *regular* constraint involves:

- an array of integer variables *VARS* and
- a deterministic finite automaton *AUTOMATON*.

It enforces the sequences of *VARS* to be a word recognized by *AUTOMATON*.

There are various ways to declare the automaton:

- create a `FiniteAutomaton` and add states, initial and final ones and transitions (see `FiniteAutomaton` API for more details),
- create a `FiniteAutomaton` with a regexp as argument.

Beware: `FiniteAutomaton` only handles values between 0 and 65535, because it relies on `java.Character`.

Implementation based on: [Pes04].

API:

```
Constraint regular(IntVar[] VARS, IAutomaton AUTOMATON)
```

Example

```
1        Solver solver = new Solver();
2        IntVar[] CS = VF.enumeratedArray("CS", 4, 1, 5, solver);
3        solver.post(ICF.regular(CS,
4                new FiniteAutomaton("(1|2)(3*)(4|5)")));
5        solver.findAllSolutions();
```

The solutions of the problem are :

- *CS[0] = 1, CS[1] = 3, CS[2] = 3, CS[3] = 4*
- *CS[0] = 1, CS[1] = 3, CS[2] = 3, CS[3] = 5*
- *CS[0] = 2, CS[1] = 3, CS[2] = 3, CS[3] = 4*
- *CS[0] = 2, CS[1] = 3, CS[2] = 3, CS[3] = 5*

20.41 scalar

The *scalar* constraint involves:

- an array of integer variables *VARS*,
- an array of integer *COEFFS*,
- an optional operator *OPERATOR* and
- an integer variable *SCALAR*.

It ensures that *sum(VARS[i]*COEFFS[i]) OPERATOR SCALAR*; where *OPERATOR* must be chosen from `{"=", "!=", ">","<",">=","<="}`. The *scalar* constraint filters on bounds only. The constraint suppress variables with coefficients set to 0, recognizes *sum* (when all coefficients are equal to *-1*, or all equal to *-1*), and enables, under certain conditions, to reformulate the constraint with a *table* constraint providint AC filtering algorithm.

See also: scalar_product in the Global Constraint Catalog.

Implementation based on: [HS02].

API:

```
Constraint scalar(IntVar[] VARS, int[] COEFFS, IntVar SCALAR)
Constraint scalar(IntVar[] VARS, int[] COEFFS, String OPERATOR, IntVar SCALAR)
```

Example

```
        Solver solver = new Solver();
        IntVar[] CS = VF.enumeratedArray("CS", 4, 1, 4, solver);
        int[] coeffs = new int[]{1, 2, 3, 4};
        IntVar R = VF.bounded("R", 0, 20, solver);
        solver.post(ICF.scalar(CS, coeffs, R));
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *CS[0] = 1, CS[1] = 1, CS[2] = 1, CS[3] = 1, R = 10*
- *CS[0] = 1, CS[1] = 2, CS[2] = 3, CS[3] = 1, R = 18*
- *CS[0] = 1, CS[1] = 4, CS[2] = 2, CS[3] = 1, R = 19*
- *CS[0] = 1, CS[1] = 2, CS[2] = 1, CS[3] = 3, R = 20*

20.42 sort

The *sort* constraint involves two arrays of integer variables *VARS* and *SORTEDVARS*. It ensures that the variables of *SORTEDVARS* correspond to the variables of *VARS* according to a permutation. Moreover, the variable of *SORTEDVARS* are sorted in increasing order.

See also: sort in the Global Constraint Catalog.

Implementation based on: [MT00].

API:

```
Constraint sort(IntVar[] VARS, IntVar[] SORTEDVARS)
```

Example

```
        Solver solver = new Solver();
        IntVar[] X = VF.enumeratedArray("X", 3, 0, 2, solver);
        IntVar[] Y = VF.enumeratedArray("Y", 3, 0, 2, solver);
        solver.post(ICF.sort(X, Y));
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *X[0] = 0, X[1] = 0, X[2] = 0, Y[0] = 0, Y[1] = 0, Y[2] = 0*
- *X[0] = 1, X[1] = 0, X[2] = 2, Y[0] = 0, Y[1] = 1, Y[2] = 2*
- *X[0] = 2, X[1] = 1, X[2] = 0, Y[0] = 0, Y[1] = 1, Y[2] = 2*
- *X[0] = 2, X[1] = 1, X[2] = 2, Y[0] = 1, Y[1] = 2, Y[2] = 2*

20.43 square

The `square` constraint involves two variables *VAR1* and *VAR2*. It ensures that *VAR1* = *VAR2*^{2}.

API:

```
Constraint square(IntVar VAR1, IntVar VAR2)
```

Example

```
        Solver solver = new Solver();
        IntVar X = VF.enumerated("X", 0, 5, solver);
        IntVar Y = VF.enumerated("Y", -1, 3, solver);
        solver.post(ICF.square(X, Y));
        solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 1, Y = -1*
- *X = 0, Y = 0*
- *X = 1, Y = 1*
- *X = 4, Y = 2*

20.44 subcircuit

The *subcircuit* constraint involves:

- an array of integer variables *VARS*,
- an integer *OFFSET* and
- an integer variable *SUBCIRCUIT_SIZE*.

It ensures that the elements of *VARS* define a single circuit of *SUBCIRCUIT_SIZE* nodes where:

- *VARS[i] = OFFSET+j* means that *j* is the successor of *i*,
- *VARS[i] = OFFSET+i* means that *i* is not part of the circuit.

It also ensures that $|\{VARS[i] \neq OFFSET+i\}| = SUBCIRCUIT\_SIZE$.

Implementation based on: *circuit*.

API:

```
Constraint subcircuit(IntVar[] VARS, int OFFSET, IntVar SUBCIRCUIT_SIZE)
```

Example

```
        Solver solver = new Solver();
        IntVar[] NODES = VF.enumeratedArray("NS", 5, 0, 4, solver);
        IntVar SI = VF.enumerated("SI", 2, 3, solver);
```

```
        solver.post(ICF.subcircuit(NODES, 0, SI));
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *NS[0] = 0, NS[1] = 1, NS[2] = 2, NS[3] = 4, NS[4] = 3, SI = 2*
- *NS[0] = 4, NS[1] = 1, NS[2] = 2, NS[3] = 3, NS[4] = 0, SI = 2*
- *NS[0] = 1, NS[1] = 2, NS[2] = 0, NS[3] = 3, NS[4] = 4, SI = 3*
- *NS[0] = 3, NS[1] = 1, NS[2] = 2, NS[3] = 4, NS[4] = 0, SI = 3*

20.45 subpath

The *subpath* constraint involves:

- an array of integer variables *VARS*,
- an integer variable *START*,
- an integer variable *END*,
- an integer *OFFSET* and
- an integer variable *SIZE*.

It ensures that the elements of *VARS* define a path of *SIZE* vertices, leading from *START* to *END* where:

- *VARS[i] = OFFSET+j* means that *j* is the successor of *i*,
- *VARS[i] = OFFSET+i* means that vertex *i* is excluded from the path.

Moreover, *VARS[END-OFFSET]* = | *VARS* | +'OFFSET'.

See also: subpath in the Global Constraint Catalog.

Implementation based on: *path, circuit*.

API:

```
Constraint[] subpath(IntVar[] VARS, IntVar START, IntVar END, int OFFSET, IntVar SIZE)
```

Example

```
        Solver solver = new Solver();
        IntVar[] VS = VF.enumeratedArray("VS", 4, 0, 4, solver);
        IntVar S = VF.enumerated("S", 0, 3, solver);
        IntVar E = VF.enumerated("E", 0, 3, solver);
        IntVar SI = VF.enumerated("SI", 2, 3, solver);
        solver.post(ICF.subpath(VS, S, E, 0, SI));
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *VS[0] = 1, VS[1] = 4, VS[2] = 2, VS[3] = 3, S = 0, E = 1, SI = 2*
- *VS[0] = 4, VS[1] = 1, VS[2] = 2, VS[3] = 0, S = 3, E = 0, SI = 2*
- *VS[0] = 3, VS[1] = 1, VS[2] = 4, VS[3] = 2, S = 0, E = 2, SI = 3*

- *VS[0] = 0, VS[1] = 2, VS[2] = 4, VS[3] = 1, S = 3, E = 2, SI = 3*

20.46 sum

The *sum* constraint involves:

- an array of integer (or boolean) variables *VARS*,
- an optional operator *OPERATOR* and
- an integer variable *SUM*.

It ensures that *sum(VARS[i]) OPERATOR SUM*; where operator must be chosen among `{"=", "!=", ">","<",">=","<="}`. If no operator is defined, `"="` is set by default. Note that when the operator differs from `"="`, an intermediate variable is declared and an *arithm* constraint is returned. For performance reasons, a specialization for boolean variables is provided.

See also: scalar_product in the Global Constraint Catalog.

Implementation based on: [HS02].

API:

```
Constraint sum(IntVar[] VARS, IntVar SUM)
Constraint sum(IntVar[] VARS, String OPERATOR, IntVar SUM)
Constraint sum(BoolVar[] VARS, IntVar SUM)
Constraint sum(BoolVar[] VARS, String OPERATOR, IntVar SUM)
```

Example

```
1        Solver solver = new Solver();
2        IntVar[] VS = VF.enumeratedArray("VS", 4, 0, 4, solver);
3        IntVar SU = VF.enumerated("SU", 2, 3, solver);
4        solver.post(ICF.sum(VS, "<=", SU));
5        solver.findAllSolutions();
```

Some solutions of the problem are :

- *VS[0] = 0 VS[1] = 0 VS[2] = 0 VS[3] = 0 SU = 2*
- *VS[0] = 0 VS[1] = 0 VS[2] = 0 VS[3] = 2 SU = 2*
- *VS[0] = 0 VS[1] = 0 VS[2] = 0 VS[3] = 3 SU = 3*
- *VS[0] = 1 VS[1] = 1 VS[2] = 0 VS[3] = 0 SU = 3*

20.47 table

The `table` constraint involves either:

- two variables *VAR1* and *VAR2*, a list of pair of values, named *TUPLES* and an algorithm *ALGORITHM*.
- or an array of variables *VARS*, a list of tuples of values, named *TUPLES* and an algorithm *ALGORITHM*.

It is an extensional constraint enforcing, most of the time, arc-consistency.

When only two variables are involved, the available algorithms are:

- `"AC2001"`: applies the AC2001 algorithm,
- `"AC3"`: applies the AC3 algorithm,
- `"AC3rm"`: applies the AC3rm algorithm,
- `"AC3bit+rm"`: (default) applies the AC3bit+rm algorithm,
- `"FC"`: applies the forward checking algorithm.

When more than two variables are involved, the available algorithms are:

- `"GAC2001"`: applies the GAC2001 algorithm,
- `"GAC2001+"`: applies the GAC2001 algorithm for allowed tuples only,
- `"GAC3rm"`: applies the GAC3 algorithm,
- `"GAC3rm+"`: (default) applies the GAC3rm algorithm for allowed tuples only,
- `"GACSTR+"`: applies the GAC version STR for allowed tuples only,
- `"STR2+"`: applies the GAC STR2 algorithm for allowed tuples only,
- `"FC"`: applies the forward checking algorithm.

Implementation based on: [tbd].

API:

```
Constraint table(IntVar VAR1, IntVar VAR2, Tuples TUPLES, String ALGORITHM)
Constraint table(IntVar[] VARS, Tuples TUPLES, String ALGORITHM)
```

Example

```
1         Solver solver = new Solver();
2         IntVar X = VF.enumerated("X", 0, 5, solver);
3         IntVar Y = VF.enumerated("Y", -1, 3, solver);
4         Tuples tuples = new Tuples(true);
5         tuples.add(1, -2);
6         tuples.add(1, 1);
7         tuples.add(4, 2);
8         tuples.add(1, 4);
9         solver.post(ICF.table(X, Y, tuples, "AC2001"));
10        solver.findAllSolutions();
```

The solutions of the problem are :

- *X = 1, Y = 1*
- *X = 4, Y = 2*

20.48 times

The *times* constraints involves either:

- three variables X, Y and Z. It ensures that $X \times Y = Z$.

- or two variables X and Z and a constant y. It ensures that $X \times y = Z$.

The propagator of the *times* constraint filters on bounds only. If the option is enabled and under certain condition, the *times* constraint may be redefined with a *table* constraint, providing a better filtering algorithm.

The API are :

```
Constraint times(IntVar X, IntVar Y, IntVar Z)
Constraint times(IntVar X, int Y, IntVar Z)
```

Example

```
        Solver solver = new Solver();
        IntVar X = VF.enumerated("X", -1, 2, solver);
        IntVar Y = VF.enumerated("Y", 2, 4, solver);
        IntVar Z = VF.enumerated("Z", 5, 7, solver);
        solver.post(ICF.times(X, Y, Z));
        solver.findAllSolutions();
```

The solution of the problem is :

- *X = 2 Y = 3 Z = 6*

20.49 tree

The *tree* constraint involves:

- an array of integer variables *SUCCS*,
- an integer variable *NBTREES* and
- an integer *OFFSET*.

It partitions the *SUCCS* variables into *NBTREES* (anti) arborescences:

- *SUCCS[i] = OFFSET+j* means that *j* is the successor of *i*,
- *SUCCS[i] = OFFSET+i* means that *i* is a root.

See also: tree in the Global Constraint Catalog.

Implementation based on: [FL11].

API:

```
Constraint tree(IntVar[] SUCCS, IntVar NBTREES, int OFFSET)
```

Example

```
        Solver solver = new Solver();
        IntVar[] VS = VF.enumeratedArray("VS", 4, 0, 4, solver);
        IntVar NT = VF.enumerated("NT", 2, 3, solver);
        solver.post(ICF.tree(VS, NT, 0));
```

Some solutions of the problem are :

- *VS[0] = 0, VS[1] = 1, VS[2] = 1, VS[3] = 1, NT = 2*

- *VS[0] = 1, VS[1] = 1, VS[2] = 2, VS[3] = 1, NT = 2*
- *VS[0] = 2, VS[1] = 0, VS[2] = 2, VS[3] = 3, NT = 2*
- *VS[0] = 0, VS[1] = 3, VS[2] = 2, VS[3] = 3, NT = 3*
- *VS[0] = 3, VS[1] = 1, VS[2] = 2, VS[3] = 3, NT = 3*

20.50 TRUE

The *TRUE* constraint is always satisfied. It should only be used with `LogicalFactory`.

20.51 tsp

The *tsp* constraint involves:

- an array of integer variables *SUCCS*,
- an integer variable *COST* and
- a matrix of integers *COST_MATRIX*.

It formulates the Travelling Salesman Problem: the variables *SUCCS* form a hamiltonian circuit of value *COST*. Going from *i* to *j*, *SUCCS[i] = j*, costs *COST_MATRIX[i][j]*.

This constraint is not a built-in constraint and is based on various propagators.

The filtering power of this constraint remains limited. For stronger filtering, use the *choco-graph* extension (https://github.com/chocoteam/choco-graph/releases/tag/choco-graph-3.2.1) which includes powerful cost-based filtering.

API:

```
Constraint[] tsp(IntVar[] SUCCS, IntVar COST, int[][] COST_MATRIX)
```

Example

```
1        Solver solver = new Solver();
2        IntVar[] VS = VF.enumeratedArray("VS", 4, 0, 4, solver);
3        IntVar CO = VF.enumerated("CO", 0, 15, solver);
4        int[][] costs = new int[][]{{0, 1, 3, 7}, {1, 0, 1, 3}, {3, 1, 0, 1}, {7, 3, 1, 0}};
5        solver.post(ICF.tsp(VS, CO, costs));
6        solver.findAllSolutions();
```

The solutions of the problem are :

- *VS[0] = 2, VS[1] = 0, VS[2] = 3, VS[3] = 1, CO = 8*
- *VS[0] = 3, VS[1] = 0, VS[2] = 1, VS[3] = 2, CO = 10*
- *VS[0] = 1, VS[1] = 2, VS[2] = 3, VS[3] = 0, CO = 10*
- *VS[0] = 3, VS[1] = 2, VS[2] = 0, VS[3] = 1, CO = 14*
- *VS[0] = 1, VS[1] = 3, VS[2] = 0, VS[3] = 2, CO = 8*
- *VS[0] = 2, VS[1] = 3, VS[2] = 1, VS[3] = 0, CO = 14*

CHAPTER 21

Constraints over set variables

21.1 all_different

The *all_different* constraint involves an array of set variables *SETS*. It ensures that sets in *SETS* are all different (not necessarily disjoint). Note that there cannot be more than two empty sets.

API:

```
Constraint all_different(SetVar[] SETS)
```

21.2 all_disjoint

The *all_disjoint* constraint involves an array of set variables *SETS*. It ensures that all sets from *SETS* are disjoint. Note that there can be multiple empty sets.

API:

```
Constraint all_disjoint(SetVar[] SETS)
```

21.3 all_equal

The *all_equal* constraint involves an array of set variables *SETS*. It ensures that sets in *SETS* are all equal.

API:

```
Constraint all_equal(SetVar[] SETS)
```

21.4 bool_channel

The *bool_channel* constraint involves:

- an array of boolean variables *BOOLEANS*,
- a set variable *SET* and

- an integer *OFFSET*.

It channels *BOOLEANS* and *SET* such that : $i \in$ SET $\Leftrightarrow$ *BOOLEANS[i-OFFSET] = 1*.

API:

```
Constraint bool_channel(BoolVar[] BOOLEANS, SetVar SET, int OFFSET)
```

21.5 cardinality

The *cardinality* constraint involves:

- a set variable *SET* and
- an integer variable *CARD*.

It ensures that | *SET_VAR* | = *CARD*.

The API is :

```
Constraint cardinality(SetVar SET, IntVar CARD)
```

21.6 disjoint

The *disjoint* constraint involves two set variables *SET_1* and *SET_2*. It ensures that *SET_1* and *SET_2* are disjoint, that is, they cannot contain the same element. Note that they can be both empty.

API:

```
Constraint disjoint(SetVar SET_1, SetVar SET_2)
```

21.7 element

The *element* constraint involves:

- an integer variable *INDEX*,
- and array of set variables *SETS*,
- an integer *OFFSET* and
- a set variable *SET*.

It ensures that *SETS[INDEX-OFFSET] = SET*.

API:

```
Constraint element(IntVar INDEX, SetVar[] SETS, int OFFSET, SetVar SET)
```

21.8 int_channel

The *int_channel* constraint involves:

- an array of set variables *SETS*,
- an array of integer variables *INTEGERS*,
- two integers *OFFSET_1* and *OFFSET_2*.

It ensures that: $x \in SETS[y\text{-}OFFSET\_1] \Leftrightarrow INTEGERS[x\text{-}OFFSET\_2] = y$.

The API is :

```
Constraint int_channel(SetVar[] SETS, IntVar[] INTEGERS, int OFFSET_1, int OFFSET_2)
```

21.9 int_values_union

The *int_values_union* constraint involves:

- an array of integer variables *VARS* and
- a set variable *VALUES*

It ensures that: $\text{VALUES} = \text{VARS}_1 \cup \text{VARS}_2 \cup \ldots \cup \text{VARS}_n$.

The API is :

```
Constraint int_values_union(IntVar[] VARS, SetVar VALUES)
```

21.10 intersection

The *intersection* constraint involves:

- an array of set variables *SETS* and
- a set variable *INTERSECTION*.

It ensures that *INTERSECTION* is the intersection of the sets *SETS*.

The API is :

```
Constraint intersection(SetVar[] SETS, SetVar INTERSECTION)
```

21.11 inverse_set

The *inverse_set* constraint involves:

- an array of set variables *SETS*,
- an array of set variable *INVERSE_SETS* and
- two integers *OFFSET_1* and *OFFSET_2*.

It ensures that *x :math:*in‘ *SETS[y-OFFSET_1]* ⇔ y ∈ *INVERSE_SETS[x-OFFSET_2]*.

API:

```
Constraint inverse_set(SetVar[] SETS, SetVar[] INVERSE_SETS, int OFFSET_1, int OFFSET_2)
```

21.12 max

The *max* constraint involves:

- either:
  - a set variable *SET*,
  - an integer variable *MAX_ELEMENT_VALUE* and
  - a boolean *NOT_EMPTY*.

  It ensures that *MIN_ELEMENT_VALUE* is equal to the maximum element of *SET*.
- or:
  - a set variable *SET*,
  - an array of integer *WEIGHTS*,
  - an integer *OFFSET*,
  - an integer variable *MAX_ELEMENT_VALUE* and
  - a boolean *NOT_EMPTY*.

  It ensures that *max(WEIGHTS[i-OFFSET] | i in INDEXES) = MAX_ELEMENT_VALUE*.

The boolean *NOT_EMPTY* set to *true* states that *INDEXES* cannot be empty.

API:

```
Constraint max(SetVar SET, IntVar MAX_ELEMENT_VALUE, boolean NOT_EMPTY)
Constraint max(SetVar INDEXES, int[] WEIGHTS, int OFFSET, IntVar MAX_ELEMENT_VALUE, boolean NOT_EMPT
```

21.13 member

The *member* constraint involves:

- either:
  - an array of set variables *SETS* and
  - a set variable *SET*.

  It ensures that *SET* belongs to *SETS*.
- or:
  - an integer variable *INTEGER* and
  - a set variable *SET*.

  It ensures that *INTEGER* is included in *SET*.

API:

```
Constraint member(SetVar[] SETS, SetVar SET)
Constraint member(IntVar INTEGER, SetVar SET)
```

21.14 not_member

The *not_member* constraint involves:

- an integer variable *INTEGER* and
- a set variable *SET*.

It ensures that *INTEGER* is not included in *SET*.

API:

```
Constraint not_member(IntVar INTEGER, SetVar SET)
```

21.15 min

The *min* constraint involves:

- either:
  - a set variable *SET*,
  - an integer variable *MIN_ELEMENT_VALUE* and
  - a boolean *NOT_EMPTY*.

  It ensures that *MIN_ELEMENT_VALUE* is equal to the minimum element of *SET*.
- or:
  - a set variable *SET*,
  - an array of integer *WEIGHTS*,
  - an integer *OFFSET*,
  - an integer variable *MAX_ELEMENT_VALUE* and
  - a boolean *NOT_EMPTY*.

  It ensures that *min(WEIGHTS[i-OFFSET] | i in INDEXES) = MIN_ELEMENT_VALUE*.

The boolean *NOT_EMPTY* set to *true* states that *INDEXES* cannot be empty.

API:

```
Constraint min(SetVar SET, IntVar MIN_ELEMENT_VALUE, boolean NOT_EMPTY)
Constraint min(SetVar INDEXES, int[] WEIGHTS, int OFFSET, IntVar MIN_ELEMENT_VALUE, boolean NOT_EMPT
```

21.16 nbEmpty

The *nbEmpty* constraint involves:

- an array of set variables *SETS* and
- an integer variable *NB_EMPTY_SETS*.

It restricts the number of empty sets in *SETS* to be equal *NB_EMPTY_SET*.

API:

```
Constraint nbEmpty(SetVar[] SETS, IntVar NB_EMPTY_SETS)
```

21.17 notEmpty

The *notEmpty* constraint involves a set variable *SET*.

It prevents *SET* to be empty.

API:

```
Constraint notEmpty(SetVar SET)
```

21.18 offSet

The *offset* constraint involves:

- two set variables *SET_1* and *SET_2* and
- an integer *OFFSET*.

It ensures that to any value *x* in *SET_1*, the value *x+OFFSET* is in *SET_2* (and reciprocally).

API:

```
Constraint offSet(SetVar SET_1, SetVar SET_2, int OFFSET)
```

21.19 partition

The *partition* constraint involves:

- an array of set variables *SETS* and
- a set variable *UNIVERSE*.

It ensures that *UNVIVERSE* is partitioned in disjoint sets *SETS*.

API:

```
Constraint partition(SetVar[] SETS, SetVar UNIVERSE)
```

21.20 subsetEq

The *subsetEq* constraint involves an array of set variables *SETS*. It ensures that $i<j \Leftrightarrow SET\_VARS[i] \subseteq SET\_VARS[j]$.

The API is :

```
Constraint subsetEq(SetVar[] SETS)
```

21.21 sum

The *sum* constraint involves:

- a set variables *INDEXES*,
- an array of integer *WEIGHTS*,
- an integer *OFFSET*,
- an integer variable *SUM* and
- a boolean *NOT_EMPTY*.

The constraint ensures that *sum(WEIGHTS[i-OFFSET] | i in INDEXES) = SUM*. The boolean *NOT_EMPTY* set to *true* states that *INDEXES* cannot be empty.

API:

```
Constraint sum(SetVar INDEXES, int[] WEIGHTS, int OFFSET, IntVar SUM, boolean NOT_EMPTY)
```

21.22 symmetric

The *symmetric* constraint involves:

- an array of set variables *SETS* and
- an integer *OFFSET*.

It ensures that: $x \in SETS[y\text{-}OFFSET] \Leftrightarrow y \in SETS[x\text{-}OFFSET]$.

API:

```
Constraint symmetric(SetVar[] SETS, int OFFSET)
```

21.23 union

The *union* constraint involves:

- an array of set variables *SETS* and
- a set variable *UNION*.

It ensures that *SET_UNION* is equal to the union if the sets in *SET_VARS*.

The API is :

```
Constraint union(SetVar[] SETS, SetVar UNION)
```

CHAPTER 22

Constraints over real variables

Real constraints are managed externally with *Ibex*. Due to the limited number of declaration possibilities, there is no factory for real constraints. Indeed, posting a `RealConstraint` is enough.

The available constructors are:

```
RealConstraint(String name, String functions, int option, RealVar... rvars)
RealConstraint(String name, String functions, RealVar... rvars)
RealConstraint(String functions, RealVar... rvars)
```

- `name` enables to set a name to the constraint.
- `functions` is a `String` which defines the list of functions to hold, separated with semi-colon ";".

A function is a declared using the following format:

- the '{i}' tag defines a variable, where 'i' is an explicit index the array of variables `rvars`,
- one or more operators :'+,-,*,/,=,<,>,<=,>=,exp( ),ln( ),max( ),min( ),abs( ),cos( ), sin( ),...'

A complete list is available in the documentation of IBEX. - `rvars` is the list of involved real variables. - `option` is enable to state the propagation option (default is `Ibex.COMPO`).

Example

```
1        double PREC = 0.01d; // precision
2        RealVar x = VariableFactory.real("x", -1.0d, 1.0d, PREC, solver);
3        RealVar y = VariableFactory.real("y", -1.0d, 1.0d, PREC, solver);
4        RealConstraint rc = new RealConstraint(
5                "my fct",
6                "({0}*{1})+sin({0})=1.0;ln({0}+[-0.1,0.1])>=2.6",
7                Ibex.HC4,
8                x, y);
9        solver.post(rc);
10       Chatterbox.showSolutions(solver);
11   }
```

CHAPTER 23

Sat solver

23.1 addAtMostNMinusOne

Add a clause to the SAT constraint whic states that: $BOOLVARS_1 + BOOLVARS_2 + ... + BOOLVARS_n < |BOOLVARS|$.

API:

```
boolean addAtMostNMinusOne(BoolVar[] BOOLVARS)
```

Example

```
1         Solver solver = new Solver();
2         BoolVar[] BVARS = VF.boolArray("BS", 4, solver);
3         SatFactory.addAtMostNMinusOne(BVARS);
4         solver.findAllSolutions();
```

Some solutions of the problem are :

- *BS[0] = 1, BS[1] = 1, BS[2] = 1, BS[3] = 0*
- *BS[0] = 1, BS[1] = 0, BS[2] = 1, BS[3] = 0*
- *BS[0] = 0, BS[1] = 1, BS[2] = 1, BS[3] = 1*
- *BS[0] = 0, BS[1] = 0, BS[2] = 0, BS[3] = 1*
- *BS[0] = 0, BS[1] = 0, BS[2] = 0, BS[3] = 0*

23.2 addAtMostOne

Add a clause to the SAT constraint whic states that: $BOOLVARS_1 + BOOLVARS_2 + ... + BOOLVARS_n \leq 1$.

API:

```
boolean addAtMostOne(BoolVar[] BOOLVARS)
```

Example

```
        Solver solver = new Solver();
        BoolVar[] BVARS = VF.boolArray("BS", 4, solver);
        SatFactory.addAtMostOne(BVARS);
        solver.findAllSolutions();
```

The solutions of the problem are :

- *BS[0] = 1, BS[1] = 0, BS[2] = 0, BS[3] = 0*
- *BS[0] = 0, BS[1] = 1, BS[2] = 0, BS[3] = 0*
- *BS[0] = 0, BS[1] = 0, BS[2] = 1, BS[3] = 0*
- *BS[0] = 0, BS[1] = 0, BS[2] = 0, BS[3] = 1*
- *BS[0] = 0, BS[1] = 0, BS[2] = 0, BS[3] = 0*

23.3 addBoolAndArrayEqualFalse

Add a clause to the SAT constraint whic states that: $|not`(`BOOLVARS_1 \wedge BOOLVARS_2 \wedge ... \wedge BOOLVARS_n)$.

API:

boolean addBoolAndArrayEqualFalse(BoolVar[] BOOLVARS)

Example

```
        Solver solver = new Solver();
        BoolVar[] BVARS = VF.boolArray("BS", 4, solver);
        SatFactory.addBoolAndArrayEqualFalse(BVARS);
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *BS[0] = 1, BS[1] = 1, BS[2] = 1, BS[3] = 0*
- *BS[0] = 1, BS[1] = 0, BS[2] = 1, BS[3] = 1*
- *BS[0] = 1, BS[1] = 0, BS[2] = 0, BS[3] = 0*
- *BS[0] = 0, BS[1] = 1, BS[2] = 0, BS[3] = 1*
- *BS[0] = 0, BS[1] = 0, BS[2] = 0, BS[3] = 0*

23.4 addBoolAndArrayEqVar

Add a clause to the SAT constraint which states that: $(BOOLVARS_1 \wedge BOOLVARS_2 \wedge ... \wedge BOOLVARS_n) \Leftrightarrow TARGET$.

API:

```
boolean addBoolAndArrayEqVar(BoolVar[] BOOLVARS, BoolVar TARGET)
```

Example

```
        Solver solver = new Solver();
        BoolVar[] BVARS = VF.boolArray("BS", 4, solver);
        BoolVar T = VF.bool("T", solver);
        SatFactory.addBoolAndArrayEqVar(BVARS, T);
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *BS[0] = 1, BS[1] = 1, BS[2] = 1, BS[3] = 1 T = 1*
- *BS[0] = 1, BS[1] = 1, BS[2] = 0, BS[3] = 1, T = 0*
- *BS[0] = 0, BS[1] = 1, BS[2] = 0, BS[3] = 0, T = 0*
- *BS[0] = 0, BS[1] = 0, BS[2] = 0, BS[3] = 0, T = 0*

23.5 addBoolAndEqVar

Add a clause to the SAT constraint which states that: $(LEFT \wedge RIGTH) \Leftrightarrow TARGET$.

API:

```
boolean addBoolAndEqVar(BoolVar LEFT, BoolVar RIGHT, BoolVar TARGET)
```

Example

```
        Solver solver = new Solver();
        BoolVar L = VF.bool("L", solver);
        BoolVar R = VF.bool("R", solver);
        BoolVar T = VF.bool("T", solver);
        SatFactory.addBoolAndEqVar(L, R, T);
        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 1, R = 1, T = 1*
- *L = 1, R = 0, T = 0*
- *L = 0, R = 1, T = 0*
- *L = 0, R = 0, T = 0*

23.6 addBoolEq

Add a clause to the SAT constraint which states that the two boolean variables *LEFT* and *RIGHT* are equal.

API:

```
boolean addBoolEq(BoolVar LEFT, BoolVar RIGHT)
```

Example

```
        Solver solver = new Solver();
        BoolVar L = VF.bool("L", solver);
        BoolVar R = VF.bool("R", solver);
        SatFactory.addBoolEq(L, R);
        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 1, R = 1*
- *L = 0, R = 0*

23.7 addBoolIsEqVar

Add a clause to the SAT constraint which states that: $(LEFT = RIGTH) \Leftrightarrow TARGET$.

API:

```
boolean addBoolIsEqVar(BoolVar LEFT, BoolVar RIGHT, BoolVar TARGET)
```

Example

```
        Solver solver = new Solver();
        BoolVar L = VF.bool("L", solver);
        BoolVar R = VF.bool("R", solver);
        BoolVar T = VF.bool("T", solver);
        SatFactory.addBoolIsEqVar(L, R, T);
        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 1, R = 1, T = 1*
- *L = 1, R = 0, T = 0*
- *L = 0, R = 1, T = 0*
- *L = 0, R = 0, T = 1*

23.8 addBoolIsLeVar

Add a clause to the SAT constraint which states that: $(LEFT \leq RIGTH) \Leftrightarrow TARGET$.

API:

```
boolean addBoolIsLeVar(BoolVar LEFT, BoolVar RIGHT, BoolVar TARGET)
```

Example

```
        Solver solver = new Solver();
        BoolVar L = VF.bool("L", solver);
        BoolVar R = VF.bool("R", solver);
        BoolVar T = VF.bool("T", solver);
        SatFactory.addBoolIsLeVar(L, R, T);
        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 1, R = 1, T = 1*
- *L = 1, R = 0, T = 0*
- *L = 0, R = 1, T = 1*
- *L = 0, R = 0, T = 1*

23.9 addBoolIsLtVar

Add a clause to the SAT constraint which states that: $(LEFT < RIGTH) \Leftrightarrow TARGET$.

API:

```
boolean addBoolIsLtVar(BoolVar LEFT, BoolVar RIGHT, BoolVar TARGET)
```

Example

```
        Solver solver = new Solver();
        BoolVar L = VF.bool("L", solver);
        BoolVar R = VF.bool("R", solver);
        BoolVar T = VF.bool("T", solver);
        SatFactory.addBoolIsLtVar(L, R, T);
        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 1, R = 1, T = 0*
- *L = 1, R = 0, T = 0*
- *L = 0, R = 1, T = 1*
- *L = 0, R = 0, T = 0*

23.10 addBoolIsNeqVar

Add a clause to the SAT constraint which states that: $(LEFT \neq RIGTH) \Leftrightarrow TARGET$.

API:

```
boolean addBoolIsNeqVar(BoolVar LEFT, BoolVar RIGHT, BoolVar TARGET)
```

Example

```
        Solver solver = new Solver();
        BoolVar L = VF.bool("L", solver);
        BoolVar R = VF.bool("R", solver);
        BoolVar T = VF.bool("T", solver);
        SatFactory.addBoolIsNeqVar(L, R, T);
        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 1, R = 1, T = 0*
- *L = 1, R = 0, T = 1*
- *L = 0, R = 1, T = 1*
- *L = 0, R = 0, T = 0*

23.11 addBoolLe

Add a clause to the SAT constraint which states that the boolean variable *LEFT* is less or equal than the boolean variable *RIGHT*.

API:

```
boolean addBoolLe(BoolVar LEFT, BoolVar RIGHT)
```

Example

```
        Solver solver = new Solver();
        BoolVar L = VF.bool("L", solver);
        BoolVar R = VF.bool("R", solver);
        SatFactory.addBoolLe(L, R);
        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 1, R = 1*
- *L = 0, R = 1*
- *L = 0, R = 0*

23.12 addBoolLt

Add a clause to the SAT constraint which states that the boolean variable *LEFT* is less than the boolean variable *RIGHT*.

API:

```
boolean addBoolLt(BoolVar LEFT, BoolVar RIGHT)
```

Example

```
1        Solver solver = new Solver();
2        BoolVar L = VF.bool("L", solver);
3        BoolVar R = VF.bool("R", solver);
4        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 0, R = 1*

23.13 addBoolNot

Add a clause to the SAT constraint which states that the two boolean variables *LEFT* and *RIGHT* are not equal.

API:

```
boolean addBoolNot(BoolVar LEFT, BoolVar RIGHT)
```

Example

```
1        Solver solver = new Solver();
2        BoolVar L = VF.bool("L", solver);
3        BoolVar R = VF.bool("R", solver);
4        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 1, R = 0*
- *L = 0, R = 1*

23.14 addBoolOrArrayEqualTrue

Add a clause to the SAT constraint which states that: $BOOLVARS_1 \vee BOOLVARS_2 \vee ... \vee BOOLVARS_n$.

API:

```
boolean addBoolOrArrayEqualTrue(BoolVar[] BOOLVARS)
```

Example

```
1        Solver solver = new Solver();
2        BoolVar[] BVARS = VF.boolArray("BS", 4, solver);
3        SatFactory.addBoolOrArrayEqualTrue(BVARS);
4        solver.findAllSolutions();
```

Some solutions of the problem are :

- *BS[0] = 1, BS[1] = 1, BS[2] = 1, BS[3] = 1*
- *BS[0] = 1, BS[1] = 1, BS[2] = 0, BS[3] = 0*
- *BS[0] = 1, BS[1] = 0, BS[2] = 0, BS[3] = 0*
- *BS[0] = 0, BS[1] = 1, BS[2] = 0, BS[3] = 0*
- *BS[0] = 0, BS[1] = 0, BS[2] = 0, BS[3] = 1*

23.15 addBoolOrArrayEqVar

Add a clause to the SAT constraint which states that: $(BOOLVARS_1 \vee BOOLVARS_2 \vee ... \vee BOOLVARS_n) \Leftrightarrow TARGET$.

API:

```
boolean addBoolOrArrayEqVar(BoolVar[] BOOLVARS, BoolVar TARGET)
```

Example

```
1        Solver solver = new Solver();
2        BoolVar[] BVARS = VF.boolArray("BS", 4, solver);
3        BoolVar T = VF.bool("T", solver);
4        SatFactory.addBoolOrArrayEqVar(BVARS, T);
5        solver.findAllSolutions();
```

Some solutions of the problem are :

- *BS[0] = 1, BS[1] = 1, BS[2] = 1, BS[3] = 1, T = 1*
- *BS[0] = 1, BS[1] = 1, BS[2] = 0, BS[3] = 1, T = 1*
- *BS[0] = 0, BS[1] = 1, BS[2] = 0, BS[3] = 0, T = 1*
- *BS[0] = 0, BS[1] = 0, BS[2] = 0, BS[3] = 0, T = 0*

23.16 addBoolOrEqVar

Add a clause to the SAT constraint which states that: $(LEFT \vee RIGTH) \Leftrightarrow TARGET$.

API:

```
boolean addBoolOrEqVar(BoolVar LEFT, BoolVar RIGHT, BoolVar TARGET)
```

Example

```
1        Solver solver = new Solver();
2        BoolVar L = VF.bool("L", solver);
3        BoolVar R = VF.bool("R", solver);
4        BoolVar T = VF.bool("T", solver);
```

```
        SatFactory.addBoolOrEqVar(L, R, T);
        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 1, R = 1, T = 1*
- *L = 1, R = 0, T = 1*
- *L = 0, R = 1, T = 1*
- *L = 0, R = 0, T = 0*

23.17 addBoolXorEqVar

Add a clause to the SAT constraint which states that: $(LEFT \oplus RIGTH) \Leftrightarrow TARGET$.

API:

```
boolean addBoolXorEqVar(BoolVar LEFT, BoolVar RIGHT, BoolVar TARGET)
```

Example

```
        Solver solver = new Solver();
        BoolVar L = VF.bool("L", solver);
        BoolVar R = VF.bool("R", solver);
        BoolVar T = VF.bool("T", solver);
        SatFactory.addBoolXorEqVar(L, R, T);
        solver.findAllSolutions();
```

The solutions of the problem are :

- *L = 1, R = 1, T = 0*
- *L = 1, R = 0, T = 1*
- *L = 0, R = 1, T = 1*
- *L = 0, R = 0, T = 0*

23.18 addClauses

Adding a clause involved either:

- a logical operator *TREE* and an instance of the solver,
- or, two arrays of boolean variables.

The two methods add clauses to the SAT constraint.

- The first method adds one or more clauses defined by a `LogOp`. `LopOp` aims at simplifying the declaration of clauses by providing some static methods. However, it should be considered as a last resort, due to the verbosity it comes with.

- The second API add one or more clauses defined by two arrays *POSLITS* and *NEGLITS*. The first array declares positive boolean variables, those who should be satisfied; the second array declares negative boolean variables, those who should not be satisfied.

API:

```
boolean addClauses(LogOp TREE, Solver SOLVER)
boolean addClauses(BoolVar[] POSLITS, BoolVar[] NEGLITS)
```

Example 1

```
        Solver solver = new Solver();
        BoolVar C1 = VF.bool("C1", solver);
        BoolVar C2 = VF.bool("C2", solver);
        BoolVar R = VF.bool("R", solver);
        BoolVar AR = VF.bool("AR", solver);
        SatFactory.addClauses(
                LogOp.ifThenElse(LogOp.nand(C1, C2), R, AR),
                solver);
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *C1 = 1, C2 = 0, R = 1, AR = 1*
- *C1 = 1, C2 = 0, R = 0, AR = 1*
- *C1 = 0, C2 = 1, R = 1, AR = 0*
- *C1 = 0, C2 = 0, R = 0, AR = 1*

Example 2

```
        Solver solver = new Solver();
        BoolVar P1 = VF.bool("P1", solver);
        BoolVar P2 = VF.bool("P2", solver);
        BoolVar P3 = VF.bool("P3", solver);
        BoolVar N = VF.bool("N", solver);
        SatFactory.addClauses(new BoolVar[]{P1, P2, P3}, new BoolVar[]{N});
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *P1 = 1, P2 = 1, P3 = 1, N = 1*
- *P1 = 1, P2 = 1, P3 = 1, N = 0*
- *P1 = 1, P2 = 0, P3 = 1, N = 0*
- *P1 = 0, P2 = 0, P3 = 1, N = 1*

23.19 addFalse

Add a unit clause to the SAT constraint which states that the boolean variable *BOOLVAR* must be false (equal to 0).

API:

```
boolean addFalse(BoolVar BOOLVAR)
```

Example

```
1        Solver solver = new Solver();
2        BoolVar B = VF.bool("B", solver);
3        SatFactory.addFalse(B);
4        solver.findAllSolutions();
```

The solution of the problem is :

- *B = 0*

23.20 addMaxBoolArrayLessEqVar

Add a clause to the SAT constraint which states that: maximum($BOOLVARS_i$) $\leq$ *TARGET*.

API:

```
boolean addMaxBoolArrayLessEqVar(BoolVar[] BOOLVARS, BoolVar TARGET)
```

Example

```
1        Solver solver = new Solver();
2        BoolVar[] BVARS = VF.boolArray("BS", 3, solver);
3        BoolVar T = VF.bool("T", solver);
4        SatFactory.addMaxBoolArrayLessEqVar(BVARS, T);
5        solver.findAllSolutions();
```

Some solutions of the problem are :

- *BS[0] = 1, BS[1] = 1, BS[2] = 1, T = 1*
- *BS[0] = 1, BS[1] = 0, BS[2] = 1, T = 1*
- *BS[0] = 0, BS[1] = 1, BS[2] = 1, T = 1*
- *BS[0] = 0, BS[1] = 0, BS[2] = 0, T = 0*

23.21 addSumBoolArrayGreaterEqVar

Add a clause to the SAT constraint which states that: sum($BOOLVARS_i$) $\geq$ *TARGET*.

API:

```
boolean addSumBoolArrayGreaterEqVar(BoolVar[] BOOLVARS, BoolVar TARGET)
```

Example

```
        Solver solver = new Solver();
        BoolVar[] BVARS = VF.boolArray("BS", 3, solver);
        BoolVar T = VF.bool("T", solver);
        SatFactory.addSumBoolArrayGreaterEqVar(BVARS, T);
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *BS[0] = 1, BS[1] = 1, BS[2] = 1, T = 1*
- *BS[0] = 1, BS[1] = 0, BS[2] = 1, T = 1*
- *BS[0] = 0, BS[1] = 1, BS[2] = 1, T = 1*
- *BS[0] = 0, BS[1] = 0, BS[2] = 0, T = 0*

23.22 addSumBoolArrayLessEqVar

Add a clause to the SAT constraint which states that: sum($BOOLVARS_i$) $\leq$ *TARGET*.

API:

```
boolean addSumBoolArrayLessEqVar(BoolVar[] BOOLVARS, BoolVar TARGET)
```

Example

```
        Solver solver = new Solver();
        BoolVar[] BVARS = VF.boolArray("BS", 3, solver);
        BoolVar T = VF.bool("T", solver);
        SatFactory.addSumBoolArrayLessEqVar(BVARS, T);
        solver.findAllSolutions();
```

Some solutions of the problem are :

- *BS[0] = 1 BS[1] = 1 BS[2] = 1 T = 1*
- *BS[0] = 1 BS[1] = 0 BS[2] = 1 T = 1*
- *BS[0] = 0 BS[1] = 1 BS[2] = 1 T = 1*
- *BS[0] = 0 BS[1] = 0 BS[2] = 0 T = 1*

23.23 addTrue

Add a unit clause to the SAT constraint which states that the boolean variable *BOOLVAR* must be true (equal to 1).

API:

```
boolean addTrue(BoolVar BOOLVAR)
```

Example

```
        Solver solver = new Solver();
        BoolVar B = VF.bool("B", solver);
        SatFactory.addTrue(B);
        solver.findAllSolutions();
```

The solution of the problem is :

- *B = 1*

CHAPTER 24

Logical constraints

The `LogicalConstraintFactory` (or `LCF`) provides various interesting constraints to manipulate other constraints. These constraints are based on the concept of reification. We say a constraint `C` is reified with a boolean variable `b` when we maintain the equivalence betwen `b` being equal to true and `C` being satisfied. This means the `C` constraint may be not satisfied, hence it should not be posted to the solver.

Note: It is highly recommended to use that factory only when `SatFactory` does not meet requirements.

24.1 and

Creates a logical AND constraint on the input constraints. Each constraint is first reified and then the logical AND is made over the reified variables (a sum constraint is returned).

Alternatively, the AND constraint can be directly declared with an array of boolean variables as input.

24.2 or

Creates a logical OR constraint on the input constraints. Each constraint is first reified and then the logical OR is made over the reified variables (a sum constraint is returned).

Alternatively, the OR constraint can be directly declared with an array of boolean variables as input.

24.3 not

Creates the opposite constraint of the input constraint.

While this works for any kind of constraint (including globals), it might be a bit naive and slow.

24.4 ifThen

Creates and automatically post a constraint ensuring that if the IF statement is true then the THEN statement must be true as well.

A statement is either a binary variable (0/1) or a reified constraint (satisfied/violated)

Note that the method returns void (you cannot reify that constraint which is automatically posted). If you wish to reify it, use `ifThen_reifiable` (whose implementation differ)

24.5 ifThenElse

Creates and automatically post a constraint ensuring that if the IF statement is true then the THEN statement must be true as well. Otherwise, the ELSE statement must be true.

A statement is either a binary variable (0/1) or a reified constraint (satisfied/violated)

Note that the method returns void (you cannot reify that constraint which is automatically posted). If you wish to reify it, use `ifThenElse_reifiable` (whose implementation differ)

24.6 reification

Creates and automatically post a constraint maintaining the equivalent between a binary variable being equal to 1 and a constraint being satisfied.

Note that the method returns void (you cannot reify that constraint which is automatically posted). If you wish to reify it, use `reification_reifiable` (whose implementation differ)

CHAPTER 25

Search loop factory

The `SearchLoopFactory` (or `SLF`) provides pre-defined methods to drive the search. By default, a `Solver` is created with a Depth-Frist Search algorithm and no learning. Those two components can be modified, though.

25.1 dfs

On a call to this method, the current search loop is equipped with a Depth-First Search algorithm with binary decisions and no learning. *The previous settings are automatically erased by the new ones.*

API:

```
void dfs(Solver aSolver, AbstractStrategy<V> aSearchStrategy)
```

Even if ''aSearchStrategy'' can be set to null, it is recommended to declare a convenient search strategy.

25.2 lds

On a call to this method, the current search loop is set with a Limited Discrepancy Search algorithm with binary decisions and no learning. *The previous settings are automatically erased by the new ones.* Using LDS is relevant when the search strategy carefully adapted to the problem treated and few decisions are wrong, bottom decisions are refuted first, within the incremental limit given by the discrepancy.

API:

```
void lds(Solver aSolver, AbstractStrategy<V> aSearchStrategy, int discrepancy)
```

The *discrepancy* parameter specifies the maximum discrepancy to allowed. In practice, it starts from 0 to *discrepancy*, with a step of 1.

Even if `aSearchStrategy` can be set to null, it is recommended to declare a convenient search strategy.

25.3 dds

On a call to this method, the current search loop is set with a Depth-bounded Discrepancy Search algorithm with binary decisions and no learning. *The previous settings are automatically erased by the new ones.* DDS is an alternative to LDS wherein top decisions are refuted first.

API:

```
void dds(Solver aSolver, AbstractStrategy<V> aSearchStrategy, int discrepancy)
```

The *discrepancy* parameter specifies the maximum discrepancy to allowed. In practice, it starts from 0 to *discrepancy*, with a step of 1.

Even if ‘‘aSearchStrategy‘‘ can be set to null, it is recommended to declare a convenient search strategy.

25.4 hbfs

On a call to this method, the current search loop is set with a Hybrid Best-First Search algorithm with binary decisions and no learning. *The previous settings are automatically erased by the new ones.* HBFS is relevant when an optimization problem is treated and a good approximation of the lower bound (resp. upper bound) in minimization (resp. maximization) can be computed. Recall that it hybrids DFS and Best-First Search, the memory consumption should be carefully monitored.

API:

```
void hbfs(Solver aSolver, AbstractStrategy<V> aSearchStrategy, double a, double b, long N)
```

a and *b* indicates the range which bounds the rate of redundantly propagated decisions. *N* is the backtrack limit allocated to each DFS try, it should be large enough to limit redundancy.

Even if ‘‘aSearchStrategy‘‘ can be set to null, it is recommended to declare a convenient search strategy.

25.5 seq

This method gives the possibility to combine many moves, considered then sequentially. When the selected Move cannot be extended (resp. repaired), the following one (wrt to the input order) is selected. *The previous settings are automatically erased by the new ones.* This is a work-in-progress and it may lead to unexpected behavior when repair() is applied.

API:

```
void seq(Solver aSolver, Move... moves)
```

Even if ‘‘aSearchStrategy‘‘ can be set to null, it is recommended to declare a convenient search strategy.

25.6 restart

Equips a `aSearchLoop` with a restart strategy. It encapsulates the current move within a restart move. Every time the `restartCriterion` is met, a restart is done, the new restart limit is updated thanks to `restartStrategy`. There will be at most `restartsLimit` restarts.

API:

```
void restart(Solver aSolver, LongCriterion restartCriterion, IRestartStrategy restartStrategy, int r
```

25.7 restartOnSolutions

Equips a `aSearchLoop` with a restart strategy triggered on solutions. It encapsulates the current move within a restart move. Every time a solution is found, a restart is done.

API:

```
restartOnSolutions(Solver aSolver)
```

25.8 lns

Equips a `aSearchLoop` with a Large Neighborhood Search move. It encapsulates the current move within a LNS move. Anytime a solution is encountered, it is recorded and serves as a basis for the `neighbor`. The `neighbor` creates a *fragment*: selects variables to freeze/unfreeze wrt the last solution found. If a fragment cannot be extended to a solution, a new one is selected by restarting the search. If a fragment induces a search space which a too big to be entirely evaluated, restarting the search can be forced using the `restartCriterion`. A fast restart strategy is often a good choice.

API:

```
void lns(Solver aSolver, INeighbor neighbor)
void lns(Solver aSolver, INeighbor neighbor, ICounter restartCounter)
```

25.9 learnCBJ

Equips a `aSearchLoop` with a Conflict-based Backjumping (CBJ) explanation strategy (learn). It backtracks up to the most recent decision involved in the explanation, and forget younger decisions. Set `nogoodsOn` to true to extract nogoods from failures. Set ‘‘userFeedbackOn‘‘to true to record the propagation in conflict (only relevant when one wants to interpret the explanation of a failure).

API:

```
void learnCBJ(Solver aSolver, boolean nogoodsOn, boolean userFeedbackOn)
```

25.10 learnDBT

Equips a `aSearchLoop` with a Dynamic-Backtracking (DBT) explanation strategy (learn). It backtracks up to most recent decision involved in the explanation, keep unrelated ones. Set `nogoodsOn` to true to extract nogoods from failures. Set ‘‘userFeedbackOn‘‘to true to record the propagation in conflict (only relevant when one wants to interpret the explanation of a failure).

API:

```
void learnDBT(Solver aSolver, boolean nogoodsOn, boolean userFeedbackOn)
```

CHAPTER 26

Variable selectors

Important: By default, in case of equalities, the variable with the smallest index in the input array is returned. Otherwise, consider using a `VariableSelectorWithTies` (See *Zoom on IntStrategy*).

26.1 lexico_var_selector

A built-in variable selector which chooses the first non-instantiated integer variable to branch on, regarding the lexicographic order.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
VariableSelector<IntVar> lexico_var_selector()
```

26.2 random_var_selector

A built-in variable selector which randomly chooses an integer variable, among non-instantiated ones, to branch on.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
VariableSelector<IntVar> random_var_selector(long SEED)
```

26.3 minDomainSize_var_selector

A built-in variable selector which chooses the non-instantiated integer variable with the smallest domain to branch on.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
VariableSelector<IntVar> minDomainSize_var_selector()
```

26.4 maxDomainSize_var_selector

A built-in variable selector which chooses the non-instantiated integer variable with the largest domain to branch on.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
VariableSelector<IntVar> maxDomainSize_var_selector()
```

26.5 maxRegret_var_selector

A built-in variable selector which chooses the non-instantiated integer variable with the largest difference between the two smallest values in its domain to branch on .

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
VariableSelector<IntVar> maxRegret_var_selector()
```

CHAPTER 27

Value selectors

27.1 min_value_selector

A built-in value selector which selects the variable lower bound.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntValueSelector min_value_selector()
```

27.2 mid_value_selector

A built-in value selector which selects the value in the variable domain closest to the mean of its current bounds. It computes the middle value of the domain. Then checks if the value is contained in the domain. If not and if `floor` is set to `true` (resp. `false`) the closest integer larger (resp. smaller) than the value is selected.

Important: *mid_value_selector* should not be used with assignment decisions over bounded variables (because the decision negation would result in no inference).

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntValueSelector mid_value_selector(boolean floor)
```

27.3 max_value_selector

A built-in value selector which selects the variable upper bound.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntValueSelector max_value_selector()
```

27.4 randomBound_value_selector

A built-in value selector which randomly selects either the lower bound or the upper bound of the variable.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntValueSelector randomBound_value_selector(long SEED)
```

27.5 random_value_selector

Selects randomly a value in the variable domain.

Important: *random_value_selector* should not be used with assignment decisions over bounded variables (because the decision negation could result in no inference).

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntValueSelector random_value_selector(long SEED)
```

CHAPTER 28

Decision operators

28.1 assign

A built-in decision operator which assigns the selected variable to the selected value. Its negation is *remove*.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
DecisionOperator<IntVar> assign()
```

28.2 remove

A built-in decision operator which removes the selected value from the selected variable domain. Its negation is *assign*.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
DecisionOperator<IntVar> remove()
```

28.3 split

A built-in decision operator which splits the selected variable domain at the selected value, that is, it updates the upper bound of the variable to the selected value. Its negation is *reverse_split* on *value + 1*.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
DecisionOperator<IntVar> split()
```

28.4 reverse_split

A built-in decision operator which splits the selected variable domain at the selected value, that is, it updates the lower bound of the variable to the selected value. Its negation is *split* on *value - 1*.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
DecisionOperator<IntVar> reverse_split()
```

CHAPTER 29

Built-in strategies

29.1 custom

To build a specific strategy based on `IntVar` or `SetVar`. A strategy is based on a variable selector, a value selector, an optional decision operator and a set of variables.

Scope: `IntVar`, `SetVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy custom(VariableSelector<IntVar> VAR_SELECTOR,
                                            IntValueSelector VAL_SELECTOR,
                                            DecisionOperator<IntVar> DEC_OPERATOR,
                                            IntVar... VARS)

IntStrategy custom(VariableSelector<IntVar> VAR_SELECTOR,
                                           IntValueSelector VAL_SELECTOR,
                                           IntVar... VARS)

SetStrategy custom(VariableSelector<SetVar> varS, SetValueSelector valS, boolean enforceFirst,
                   SetVar... sets)
```

29.2 dichtotomic

To build a specific strategy based on `IntVar`. A strategy is based on a variable selector, a boolean and a set of variables. It builds a dichotomic search strategy which selects a variable thanks to VAR_SELECTOR, selects the value closest to the middle of the domain and either removes the second half interval (if LOWERFIRST is true) or the first half interval (otherwise).

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy dichotomic(VariableSelector<IntVar> VAR_SELECTOR,
                                                boolean LOWERFIRST,
                                                IntVar... VARS)
```

29.3 once

To build a specific strategy based on `IntVar`. A strategy is based on a variable selector, a value selector, an optional decision operator and a set of variables. Unlike `custom`, it builds unary decisions, that is, decisions which can be applied but not be refuted.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy once(VariableSelector<IntVar> VAR_SELECTOR,
                                IntValueSelector VAL_SELECTOR,
                                DecisionOperator<IntVar> DEC_OPERATOR,
                                IntVar... VARS)

IntStrategy once(VariableSelector<IntVar> VAR_SELECTOR,
                                IntValueSelector VAL_SELECTOR,
                                IntVar... VARS)
```

29.4 force_first

A built-in strategy which chooses the first non-instantiated variable, regarding the lexicographic order, and forces its first smallest unfixed value to be part of the kernel.

Scope: `SetVar`

Factory: `org.chocosolver.solver.search.strategy.SetStrategyFactory`

API:

```
SetStrategy force_first(SetVar... sets)
```

29.5 force_maxDelta_first

A built-in strategy which chooses the first non-instantiated variable of maximum delta (envelope's cardinality minus kernel's cardinality) and forces its smallest unfixed value to be part of the kernel.

Scope: `SetVar`

Factory: `org.chocosolver.solver.search.strategy.SetStrategyFactory`

API:

```
SetStrategy force_maxDelta_first(SetVar... sets)
```

29.6 force_minDelta_first

A built-in strategy which chooses the first non-instantiated variable of minimum delta (envelope's cardinality minus kernel's cardinality) and forces its smallest first unfixed value to be part of the kernel.

Scope: `SetVar`

Factory: `org.chocosolver.solver.search.strategy.SetStrategyFactory`

API:

```
SetStrategy force_minDelta_first(SetVar... sets)
```

29.7 lexico_LB

A built-in strategy which chooses the first non-instantiated variable, regarding the lexicographic order, and assigns it to its lower bound.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy lexico_LB(IntVar... VARS)
```

29.8 lexico_Neq_LB

A built-in strategy which chooses the first non-instantiated variable, regarding the lexicographic order, and removes its lower bound from its domain.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy lexico_Neq_LB(IntVar... VARS)
```

29.9 lexico_Split

A built-in strategy which chooses the first non-instantiated variable, regarding the lexicographic order, and removes the second half of its domain.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy lexico_Split(IntVar... VARS)
```

29.10 lexico_UB

A built-in strategy which chooses the first non-instantiated variable, regarding the lexicographic order, and assigns it to its upper bound.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy lexico_UB(IntVar... VARS)
```

29.11 minDom_LB

A built-in strategy which chooses the first non-instantiated variable with the smallest domain size, and assigns it to its lower bound.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy minDom_LB(IntVar... VARS)
```

29.12 minDom_MidValue

A built-in strategy which chooses the first non-instantiated variable with the smallest domain size, and assigns it to the value closest to its middle of its domain. When `floor` is set to true, it selects the closest value less than or equal to the middle value. Set to false, it selects the closest value greater or equal to the middle value.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy minDom_MidValue(boolean floor, IntVar... VARS)
```

29.13 maxDom_Split

A built-in strategy which chooses the first non-instantiated variable with largest domain size, and removes the second half of its domain.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy maxDom_Split(IntVar... VARS)
```

29.14 minDom_UB

A built-in strategy which chooses the first non-instantiated variable with the smallest domain size, and assigns it to its upper bound.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy minDom_UB(IntVar... VARS)
```

29.15 maxReg_LB

A built-in strategy which chooses the first non-instantiated variable with the largest difference between the two smallest values of its domain, and assigns it to its lower bound.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy maxReg_LB(IntVar... VARS)
```

29.16 objective_bottom_up

A branching strategy over the objective variable. It is activated on the first solution, and iterates over the domain in decreasing order (upper bound first).

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
ObjectiveStrategy objective_bottom_up(IntVar OBJECTIVE)
```

29.17 objective_dichotomic

A branching strategy over the objective variable. It is activated on the first solution, and iterates over the domain in increasing order (lower bound first).

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
ObjectiveStrategy objective_dichotomic(IntVar OBJECTIVE)
```

29.18 objective_top_bottom

A branching strategy over the objective variable. It is activated on the first solution, and splits the domain into two parts, and evaluates first the lower part in case of minimization and the upper part in case of maximization.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
ObjectiveStrategy objective_dichotomic(IntVar OBJECTIVE)
```

29.19 random_bound

A built-in strategy which randomly chooses a non-instantiated variable, and assigns it to one of its bounds, randomly selected.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy random_bound(IntVar[] VARS)
IntStrategy random_bound(IntVar[] VARS, long SEED)
```

29.20 random_value

A built-in strategy which randomly chooses a non-instantiated variable, and assigns it to a randomly selected value from its domain.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
IntStrategy random_value(IntVar[] VARS)
IntStrategy random_value(IntVar[] VARS, long SEED)
```

29.21 remove_first

A built-in strategy which chooses the first unfixed variable and removes its smallest unfixed value from the envelope.

Scope: `SetVar`

Factory: `org.chocosolver.solver.search.strategy.SetStrategyFactory`

API:

```
SetStrategy remove_first(SetVar... sets)
```

29.22 sequencer

A meta strategy which applies sequentially the strategies in its scope.

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
AbstractStrategy sequencer(AbstractStrategy... strategies)
```

29.23 domOverWDeg

A black-box strategy for `IntVar` which selects the non-instantiated variable with the smallest ratio $\frac{|d(x)|}{w(x)}$, where $|d(x)|$ denotes the domain size of a variable x and $w(x)$ its weighted degree. The weighted degree of a variable sums the weight of each of the constraint it is involved in where at least 2 variables remains uninstantiated. The weight of a constraint is initialized to *1* and increased by one each time a constraint propagation fails during the search.

Implementation based on: [BHLS04].

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
AbstractStrategy<IntVar> domOverWDeg(IntVar[] VARS, long SEED, IntValueSelector VAL_SELECTOR)
AbstractStrategy<IntVar> domOverWDeg(IntVar[] VARS, long SEED) // default: min_value_selector
```

29.24 activity

A black-box strategy for `IntVar` which selects the non-instantiated variable with the largest ratio $\frac{a(x)}{|d(x)|}$, where $|d(x)|$ denotes the domain size of a variable x and $a(x)$ its activity. The activity of a variable measures how often the domain of the variable is reducing during the search. Then, the value with the least activity is selected from the domain of the variable.

Implementation based on: [MH12].

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
AbstractStrategy<IntVar> activity(IntVar[] VARS, double GAMMA, double DELTA, int ALPHA,
                                  int FORCE_SAMPLING, long SEED)
AbstractStrategy<IntVar> activity(IntVar[] VARS, long SEED) // default: 0.999d, 0.2d, 8, 1
```

29.25 impact

A black-box strategy for `IntVar` which selects the non-instantiated variable with the largest impact $\sum_{a \in d(x)} 1 - I(x = a)$, $I(x = a)$ denotes the impact of assigning the variable x to a value a from its domain $d(x)$. The impact of an assignment measures the search space reduction induced by a decision, by evaluating the size of the search before and after the application of a decision. The higher the impact, the greater the search space reduction. Then, the value with the least impact is selected from the domain of the variable. An approximation of the impacts is preprocessed.

Implementation based on: [Ref04].

Scope: `IntVar`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
AbstractStrategy<IntVar> impact(IntVar[] VARS, int ALPHA, int SPLIT, int NODEIMPACT,
                                long SEED, boolean INITONLY)
AbstractStrategy<IntVar> impact(IntVar[] VARS, long SEED) // default: 2, 3, 10, true
```

29.26 lastConflict

A composite heuristic which override the defined strategy by forcing some decisions to branch on variables involved in recent conflicts. After each conflict, the last assigned variable is selected in priority, so long as a failure occurs.

Implementation based on: [LSTV09].

Scope: `Variable`

Factory: `org.chocosolver.solver.search.strategy.IntStrategyFactory`

API:

```
AbstractStrategy lastConflict(Solver SOLVER)
AbstractStrategy lastConflict(Solver SOLVER, AbstractStrategy STRAT)
AbstractStrategy lastKConflicts(Solver SOLVER, int K, AbstractStrategy STRAT)
```

CHAPTER 30

Search Monitors

30.1 geometrical

Plug a geometrical restart strategy to the solver. It performs a search with restarts controlled by the resolution event `counter` which counts events occurring during the search. Parameter `base` indicates the maximal number of events allowed in the first search tree. Once this limit is reached, a restart occurs and the search continues until `base``*``grow` events are done, and so on. After each restart, the limit number of events is increased by the geometric factor `grow`. `limit` states the maximum number of restarts.

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void geometrical(Solver solver, int base, double grow, ICounter counter, int limit)
```

30.2 luby

Branch a luby restart strategy to the solver. It is an alternative to the geometric restart policy. It performs a search with restarts controlled by the number of resolution events counted by `counter`. The maximum number of events allowed at a given restart iteration is given by base multiplied by the Las Vegas coefficient at this iteration. The sequence of these coefficients is defined recursively on its prefix subsequences: starting from the first prefix 1, the $(k+1)^th$ prefix is the k^th prefix repeated `grow` times and immediately followed by coefficient $grow^k$.

- the first coefficients for `grow` =2: [1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,1,...]
- the first coefficients for `grow` =3 : [1, 1, 1, 3, 1, 1, 1, 3, 1, 1, 1, 3, 9,...]

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
luby(Solver solver, int base, int grow, ICounter counter, int limit)
```

30.3 limitNode

Defines a limit over the number of nodes allowed during the resolution. When the limit is reached, the resolution is stopped.

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void limitNode(Solver solver, long limit)
```

30.4 limitSolution

Defines a limit over the number of solutions allowed during the resolution. When the limit is reached, the resolution is stopped.

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void limitSolution(Solver solver, long limit)
```

30.5 limitTime

Defines a limit over the run time. When the limit is reached, the resolution is stopped. The limit can be either defined in millisecond or using a String which states the duration like "WWd XXh YYm ZZs" for example: - "1d2h3m4.5s": one day, two hours, three minutes, four seconds and 500 milliseconds<p/> - "2h30m": two hours and 30 minutes<p/> - "30.5s": 30 seconds and 500 ms<p/> - "180s": three minutes

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void limitTime(Solver solver, long limit)

void limitTime(Solver solver, String duration)
```

30.6 limitThreadTime

Defines a limit over the run time, defined in a separated thread. When the limit is reached, the resolution is stopped. The limit can be either defined in millisecond or using a String which states the duration like "WWd XXh YYm ZZs" for example: - "1d2h3m4.5s": one day, two hours, three minutes, four seconds and 500 milliseconds<p/> - "2h30m": two hours and 30 minutes<p/> - "30.5s": 30 seconds and 500 ms<p/> - "180s": three minutes

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void limitThreadTime(Solver solver, long limit)

void limitThreadTime(Solver solver, String duration)
```

convertInMilliseconds

30.7 limitFail

Defines a limit over the number of fails allowed during the resolution. When the limit is reached, the resolution is stopped.

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void limitFail(Solver solver, long limit)
```

30.8 limitBacktrack

Defines a limit over the number of backtracks allowed during the resolution. When the limit is reached, the resolution is stopped.

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void limitBacktrack(Solver solver, long limit)
```

30.9 restartAfterEachSolution

Force the resolution to restart at root node after each solution.

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void restartAfterEachSolution(Solver solver)
```

30.10 nogoodRecordingOnSolution

Record nogoods from solution, that is, anytime a solution is found, a nogood is produced to prevent from finding the same solution later during the search. An array of variables, presumably decision ones, is given as input to reduce the size of the generated nogoods.

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void nogoodRecordingOnSolution(IntVar[] vars)
```

30.11 nogoodRecordingFromRestarts

Record nogoods from restarts, that is, anytime the search restarts, one or more nogoods are produced, based on the decision path, to prevent from scanning the same sub-search tree.

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void nogoodRecordingFromRestarts(Solver solver)
```

30.12 shareBestKnownBound

A method which prepares the solvers in the list to be run in parallel. It plugs tools to share between solvers the best known bound when dealing with an optimization problem.

Factory: `org.chocosolver.solver.search.loop.monitors.SearchMonitorFactory`

API:

```
void shareBestKnownBound(List<Solver> solvers)
```

Part VI

Extensions of Choco

CHAPTER 31

IO extensions

31.1 choco-parsers

choco-parsers is an extension of Choco 3. It provides a parser for the FlatZinc language, a low-level solver input language that is the target language for MiniZinc. This module follows the flatzinc standards that are used for the annual MiniZinc challenge. It only supports integer variables. You will find it at https://github.com/chocoteam/choco-parsers

31.2 choco-gui

choco-gui is an extension of Choco 3. It provides a Graphical User Interface with various views which can be simply plugged on any Choco Solver object. You will find it at https://github.com/chocoteam/choco-gui

31.3 choco-cpviz

choco-cpviz is an extension of Choco 3 to deal with cpviz library. You will find it at https://github.com/chocoteam/choco-cpviz

CHAPTER 32

Modeling extensions

32.1 choco-graph

choco-graph is a Choco 3 module which allows to search for a graph, which may be subject to graph constraints. The domain of a graph variable G is a graph interval in the form [G_lb,G_ub]. G_lb is the graph representing vertices and edges which must belong to any single solution whereas G_ub is the graph representing vertices and edges which may belong to one solution. Therefore, any value G_v must satisfy the graph inclusion "G_lb subgraph of G_v subgraph of G_ub". One may see a strong connection with set variables. A graph variable can be subject to graph constraints to ensure global graph properties (e.g. connectedness, acyclicity) and channeling constraints to link the graph variable with some other binary, integer or set variables. The solving process consists of removing nodes and edges from G_ub and adding some others to G_lb until having G_lb = G_ub, i.e. until G gets instantiated. These operations stem from both constraint propagation and search. The benefits of graph variables stem from modeling convenience and performance.

This extension has documentation. You will find it at https://github.com/chocoteam/choco-graph

32.2 choco-geost

choco-geost is a Choco 3 module which provides the GEOST global constraint. This constraint is designed for geometrical and packing applications (see http://www.emn.fr/z-info/sdemasse/gccat/Cgeost.html). You will find it at https://github.com/chocoteam/choco-geost

32.3 choco-exppar

choco-exppar is a Choco 3 module which provides an expression parser. This enables to simplify the modeling step. You will find it at https://github.com/chocoteam/choco-exppar

Part VII

References

Bibliography

[BessiereHH+05] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Among, common and disjoint constraints. In *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2005, Uppsala, Sweden, June 20-22, 2005, Revised Selected and Invited Papers*, 29–43. 2005. URL: http://dx.doi.org/10.1007/11754602_3, doi:10.1007/11754602_3.

[BHLS04] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, 146–150. 2004.

[CB04] Hadrien Cambazard and Eric Bourreau. Conception d'une contrainte globale de chemin. In *Actes des 10e Journées nationales sur la résolution pratique de problèmes NP-complets – JNPC'04*, 107–121. Angers, France, France, 2004. URL: http://hal.archives-ouvertes.fr/hal-00448531.

[CB02] Mats Carlsson and Nicolas Beldiceanu. Arc-consistency for a chain of lexicographic ordering constraints. Technical Report, 2002.

[CL97] Yves Caseau and François Laburthe. Solving Small TSPs with Constraints. In Lee Naish, editor, *Proceedings of the 14th International Conference on Logic Programming – ICLP 1997*, 316–330. MIT Press, 1997.

[CY08] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, 509–523. 2008. URL: http://dx.doi.org/10.1007/978-3-540-85958-1_34, doi:10.1007/978-3-540-85958-1_34.

[DPR06] Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4):315–333, 2006. URL: http://dx.doi.org/10.1007/s10601-006-9003-7, doi:10.1007/s10601-006-9003-7.

[EenSorensson03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, 502–518. 2003. URL: http://dx.doi.org/10.1007/978-3-540-24605-3_37, doi:10.1007/978-3-540-24605-3_37.

[Fag14] Jean-Guillaume Fages. *On the use of graphs within constraint-programming*. PhD thesis, Ecole des Mines de Nantes, 2014. URL: http://www.a4cp.org/sites/default/files/jean-guillaume_fages_-_on_the_use_of_graphs_within_constraint-programming.pdf.

[FLapegue14] Jean-Guillaume Fages and Tanguy Lapègue. Filtering atmostnvalue with difference constraints: application to the shift minimisation personnel task scheduling problem. *Artificial Intelli-*

gence, 212(0):116 – 133, 2014. URL: http://www.sciencedirect.com/science/article/pii/S0004370214000423, doi:http://dx.doi.org/10.1016/j.artint.2014.04.001.

[FL12] Jean-Guillaume Fages and Xavier Lorca. Improving the asymmetric tsp by considering graph structure. *CoRR*, 2012. URL: http://arxiv.org/abs/1206.3437.

[FLP14] Jean-Guillaume Fages, Xavier Lorca, and Thierry Petit. Self-decomposable global constraints. In *Proceedings of the 2014 Conference on ECAI 2014: 21st European Conference on Artificial Intelligence*. IOS Press, 2014.

[FL11] Jean-Guillaume Fages and Xavier Lorca. Revisiting the tree constraint. In *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, 271–285. 2011. URL: http://dx.doi.org/10.1007/978-3-642-23786-7_22, doi:10.1007/978-3-642-23786-7_22.

[FHK+02] Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Global constraints for lexicographic orderings. In *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, 93–108. 2002. URL: http://link.springer.de/link/service/series/0558/bibs/2470/24700093.htm.

[HS02] Warwick Harvey and Joachim Schimpf. Bounds consistency techniques for long linear constraints. In *In Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems*, 39–46. 2002.

[LL04] YatChiu Law and JimmyH.M. Lee. Global constraints for integer and set value precedence. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, volume 3258 of Lecture Notes in Computer Science, pages 362–376. Springer Berlin Heidelberg, 2004.

[LSTV09] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict(s) in constraint programming. *Artif. Intell.*, 173(18):1592–1614, 2009. URL: http://dx.doi.org/10.1016/j.artint.2009.09.002, doi:10.1016/j.artint.2009.09.002.

[LopezOrtizQTvB03] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, 245–250. 2003.

[MT00] Kurt Mehlhorn and Sven Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, 306–319. 2000. URL: http://dx.doi.org/10.1007/3-540-45349-0_23, doi:10.1007/3-540-45349-0_23.

[MD09] Julien Menana and Sophie Demassey. Sequencing and counting with the multicost-regular constraint. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009, Pittsburgh, PA, USA, May 27-31, 2009, Proceedings*, 178–192. 2009. URL: http://dx.doi.org/10.1007/978-3-642-01929-6_14, doi:10.1007/978-3-642-01929-6_14.

[MH12] Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings*, 228–243. 2012. URL: http://dx.doi.org/10.1007/978-3-642-29828-8_15, doi:10.1007/978-3-642-29828-8_15.

[Pes04] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, 482–495. 2004. URL: http://dx.doi.org/10.1007/978-3-540-30201-8_36, doi:10.1007/978-3-540-30201-8_36.

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, 557–571. 2004. URL: http://dx.doi.org/10.1007/978-3-540-30201-8_41, doi:10.1007/978-3-540-30201-8_41.

[Regin94] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.*, 362–367. 1994. URL: http://www.aaai.org/Library/AAAI/1994/aaai94-055.php.

[Regin95] Jean-Charles Régin. Développement d'outils algorithmiques pour l'intelligence artificielle. In *Ph. D. Thesis*. 1995.