Choco3 Documentation

Release 3.2.0

Charles Prud’homme, Jean-Guillaume Fages

March 28, 2014

CHAPTER 1

Preliminaries

1.1 Installing Choco 3.2

Choco 3.2 is a java library based on Java 7. First of all, you need to be make sure that the right version of java is installed.

1.1.1 Update the classpath

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

```
java -cp .:choco-X.y.z.jar my.project.Main
```

where `X.y.z` is replaced by 3.2.0.

1.1.2 Which jar to select ?

We provide a zip file which contains the following files:

apidocs-3.2.0.zip Javadoc of Choco-3.2.0

choco-solver-3.1.1-jar-with-dependencies.jar An ready-to-use jar file which contains *choco-environment* and *choco-solver* artifacts and dependencies; it enable modeling and solving CP problems.

choco-solver-3.2.0-sources.jar The source of the artifacts *choco-environment* and *choco-solver*.

choco-parser-3.2.0.jar A jar file base on the artifact *choco-parser* without any dependency; it should be selected to input FlatZinc files.

choco-parser-3.2.0-jar-with-dependencies.jar A ready-to-use jar file which contains the following artifacts: *choco-environment*, *choco-solver* and *choco-parser* and the required dependencies. **This should be the default choice**.

choco-samples-3.2.0-sources.jar The source of the artifact *choco-samples* made of problems modeled with Choco.

1.1.3 As a Maven Dependency

Choco is build and managed using Maven3. To declare Choco as a dependency of your project, simply update the `pom.xml` of your project by adding the following instruction:

```
<dependency>
     <groupId>choco</groupId>
     <artifactId>choco-solver</artifactId>
     <version>X.y.z</version>
</dependency>
```

where `X.y.z` is replaced by 3.2.0.

You need to add a new repository to the list of declared ones in the `pom.xml` of your project:

```
<repository>
  <id>choco.repos</id>
  <url>http://www.emn.fr/z-info/choco-repo/mvn/repository/</url>
</repository>
```

1.1.4 Compiling sources

As a Maven-based project, Choco can be installed in a few instructions. First, run the following command:

```
mvn 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
```

1.2 Overview of Choco 3.2

The following steps should be enough to start using Choco 3.2. The minimal problem should at least contains a solver, some variables and constraints to linked them together.

To facilitate the modeling, Choco 3.2 provides factories for almost every required component of CSP and its resolution:

Factory	Description
`VariableFactory`	to create any kind of variables and views (integer, boolean, set, graph and real)
`IntConstraintFactory` `SetConstraintFactory` `GraphConstraintFactory`	to declare constraints over variables
`IntStrategyFactory` `SetStrategyFactory` `GraphStrategyFactory`	to define a specific search strategy, which can be combined together with a StrategiesSequencer object
`SearchMonitorFactory`	to enable logging resolution, setting limits and restart policies.

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 3.2 to treat this equation.

```
// 1. Create a Solver
Solver solver = new Solver("my first problem");
// 2. Create variables through the variable factory
IntVar x = VariableFactory.bounded("X", 0, 5, solver);
IntVar y = VariableFactory.bounded("Y", 0, 5, solver);
// 3. Create and post constraints by using constraint factories
solver.post(IntConstraintFactory.arithm(x, "+", y, "<", 5));
// 4. Define the search strategy
solver.set(IntStrategyFactory.inputOrder_InDomainMin(new IntVar[]{x,y}));
// 5. Launch the resolution process
solver.findSolution();
```

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 thought it is a good start point.

CHAPTER 2

Modeling with Choco

2.1 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:

```
        // 1. Create a Solver
        Solver solver = new Solver("my first problem");
        // 2. Create variables through the variable factory
        IntVar x = VariableFactory.bounded("X", 0, 5, solver);
        IntVar y = VariableFactory.bounded("Y", 0, 5, solver);
        // 3. Create and post constraints by using constraint factories
        solver.post(IntConstraintFactory.arithm(x, "+", y, "<", 5));
        // 4. Define the search strategy
        solver.set(IntStrategyFactory.inputOrder_InDomainMin(new IntVar[]{x,y}));
        // 5. Launch the resolution process
        solver.findSolution();
```

2.2 Declaring variables

Choco 3.2 includes five types of variables: `IntVar`, `BoolVar`, `SetVar`, `GraphVar` and `RealVar`. A factory is available to ease the declaration of variables: `VariableFactory`. At least, a varible requires a name and a solver to be declared in. The name is only helpful for the user, to read the results computed.

2.2.1 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.

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);
```

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);
```

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);
```

Variable's view

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 = Views.fixed(1, solver);
```

`x = y + 2`:

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

`x = -y`:

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

`x = 3*y`:

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

Views can be combined together:

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

2.2.2 Set variable

2.2.3 Graph variable

A graph variable `GV` is a kind of set variable designed to model graphs. It is defined by two graphs:

- the envelope `G_E` contains nodes/arcs that potentially figure in at least one solution,
- the kernel `G_K` contains nodes/arcs that figure in every solutions.

Initially `G_K` is empty while `G_E` is set to an initial domain. Then, decisions and filtering algorithms will remove nodes or arcs from `G_E` and add some others to `G_K`. A graph variable `GV=(G_E,G_K)` is instantiated if'and only if `G_E = G_K`.

We distinguish two kind of graphs, `DirectedGraphVar` and `UndirectedGraphVar`. Then for each kind, several data structures are available and can be found in enum `GraphType`. For instance `MATRIX` involves a bitset representation while `LINKED_LIST` involves linked lists and is much more appropriate for sparse graphs.

Reification graph

A graph variable `GV=(G_E,G_K)` can be used to model a matrix `B` of boolean variables.

- Each arc `(x,y)` corresponds to the boolean variable `B[x][y]`,
- `(x,y)` not in `G_E` => `B[x][y]` is `false`',
- `(x,y)` in `G_K` => `B[x][y]` is `true`.

This channeling is very easy to set:

```
UndirectedGraphVar GV = new UndirectedGraphVar(B.length);
// create an empty default constraint
Constraint c = ConstraintFactory.makeConstraint(solver);
// channeling between B and GV
c.addPropagator(PropagatorFactory.graphBooleanChanneling(GV,B,solver);
```

Relation graph

A graph variable `GV=(G_E,G_K)` can be used to model a binary relation `R` between a set of variables `V`.

- Each node `x` represents the variable `V[x]`. If `x` is not in `G_E` then it is not concerned by the relation `R`.
- Each arc `(x,y)` of `G_E` represents the potential application of `xRy`.
- Each arc (x,y) not in G_E represents either x(!R)y, either nothing (depending of whether !R is defined or not, like reifications and half reifications).
- Each arc (x,y) of G_K implies the application of xRy.

For instance the global constraint NValue(V,N) which ensures that variables in V take exactly N different values can be reformulated by:

```
// the meaning of an arc is the equivalence relation
GraphRelation relation = GraphRelationFactory.equivalence(V);
UndirectedGraphVar GV = new UndirectedGraphVar(V.length);
// the graph GVmust contain Ncliques
Constraint nValues = GraphConstraintFactory.nCliques(GV,N,solver);i
// channeling between V and GV
nValues.addPropagator(PropagatorFactory.graphRelationChanneling(GV,V,R,solver);
```

The good thing is that such a model remains valid en case of vectorial variables (`NVector` constraint).

Relation graphs can be seen as a kind of reification but they require only 1 graph variable and $O(1)$ propagators running in $O(n^2)$ time, whereas a reified approach would imply n^2 boolean variables and propagators. Moreover, relation graphs treat the problem globally through graph theory's algorithms.

2.2.4 Real variable

Real variables have a specific status in Choco 3.2. 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", 0.2, 1.0e8, precision, solver);
```

```
        solver = new Solver("Grocery");
        double epsilon = 0.000001d;
        // 4 integer variables (price in cents)
        itemCost = VariableFactory.boundedArray("item", 4, 1, 711, solver);
        // views as real variables to be used by Ibex
        realitemCost = VariableFactory.real(itemCost, epsilon);

                solver.post(new RealConstraint("Sum",                "{0} + {1} + {2} + {3} = 711", I
                solver.post(new RealConstraint("Product",        "{0} * {1}/100 * {2}/100 * {3}/100 =
        // symmetry breaking
        solver.post(new RealConstraint("SymmetryBreaking","{0} <= {1};{1} <= {2};{2} <= {3}", Ibex.H
        solver.set(IntStrategyFactory.inputOrder_InDomainMax(itemCost));
        solver.findSolution();
        solver.getIbex().release();
```

2.3 Posting constraints

Constraints define restrictions over variables that must be respected in order to get a feasible solution. A Constraint contains several propagators that will perform filtering over variables' domains and may define its specific checker through the method `isSatisfied()`.

For instance, 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:

```
solver.post(IntConstraintFactory.sum(atLeast, sum));
```

To be effective, this constraint must be declared in the solver. This achieves using the mehtod:

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

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.

CHAPTER 3

Solving problems

3.1 Finding solutions

St

todo

todo

todo

todo

CHAPTER 4

Advanced usage

4.1 Overview of Choco 3.2

todo

todo

todo

todo

todo

CHAPTER 5

Inside Choco

todo

todo

todo

todo

CHAPTER 6

Appendix

todo

todo

todo

CHAPTER 7

Frequently Asked Questions

CHAPTER 8

Glossary

solver A solver is the central concept of the library.

CHAPTER 9

Indices and tables

- *genindex*
- *search*

Index