Choco3 Documentation

Release 3.2.0

Charles Prud'homme, Jean-Guillaume Fages

May 27, 2014

Contents

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

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-3.2.0-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	Shortcut	Enables to create
`VariableFactory`	VF	Variables and views (integer, boolean, set, graph and real)
`IntConstraintFactory`	ICF	Constraints over variables
`SetConstraintFactory`	SCF	
`GraphConstraintFactory`	GCF	
`LogicalConstraintFactory`	LCF	(Manages constraint reification)
`IntStrategyFactory`	ISF	Custom or black-box search strategies
`SetStrategyFactory`	SSF	
`GraphStrategyFactory`	GSF	
`SearchMonitorFactory`	SMF	log, 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.

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.

```java
        // 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.lexico_LB(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 think 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.lexico_LB(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` (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.

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);
```

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.

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 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 = 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

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 solutions.

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);
```

For instance, the following example imposes three set variables (`x`, `y` and `z`) to form a partition of another set variable (`universe`), whose sum of integers must be minimized, while remaining in $[\![12, 19]\!]$.

while minimizing the sum of integers in the universe variable.

```
                solver = new Solver("set union sample");
                // x initial domain
                int[] x_envelope = new int[]{1,3,2,8}; // not necessarily ordered
                int[] x_kernel = new int[]{1};
                x = VariableFactory.set("x", x_envelope, x_kernel, solver);
                // y initial domain
                int[] y_envelope = new int[]{2,6,7};
                y = VariableFactory.set("y", y_envelope, solver);
                // 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);
                // universe initial domain (note that the universe is a variable)
                int[] universe_envelope = new int[]{1,2,3,5,7,8,42};
                universe = VariableFactory.set("universe", universe_envelope, solver);
                // sum variable
                sum = VariableFactory.bounded("sum of universe", 12, 19, solver);
                // partition constraint
                solver.post(SetConstraintsFactory.partition(new SetVar[]{x, y, z}, universe));
                // restricts the sum of elements in universe
                solver.post(SetConstraintsFactory.sum(universe, sum, true));
                // set a search strategy
                solver.set(SetStrategyFactory.force_first(x, y, z, universe));
                // find the optimum
                solver.findOptimalSolution(ResolutionPolicy.MINIMIZE, sum);
```

2.2.3 Graph variable

A graph variable `GV` is a kind of set variable designed to model graphs. Its domain is defined by a graph interval: `[G_E,G_K]`

- the envelope `G_E` is a graph object which contains nodes/arcs that potentially figure in at least one solution,
- the kernel `G_K` is a graph object which 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 graph variables, `DirectedGraphVar` and `UndirectedGraphVar`. Then for each kind, several data structures are available and can be found in enum `GraphType`. For instance `BITSET` involves a bitset representation while `LINKED_LIST` involves linked lists and is much more appropriate for sparse graphs.

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", Ibex.COMPO, realitemCost
        solver.post(new RealConstraint("Product", "{0} * {1}/100 * {2}/100 * {3}/100 = 711", Ibex.HC4
        // symmetry breaking
        solver.post(new RealConstraint("SymmetryBreaking","{0} <= {1};{1} <= {2};{2} <= {3}", Ibex.HC
        solver.set(IntStrategyFactory.lexico_UB(itemCost));
        solver.findSolution();
        solver.getIbex().release();
```

2.3 Declaring 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));
```

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.

A constraint should either be posted or be reified.

2.3.1 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.

2.3.2 Reifying constraints

In Choco 3.2, it is possible to reify any constraint. Reifying a constraint means associating it with a `BoolVar` to represent whether the constraint holds or not:

```
constraint.reifyWith(BoolVar b);
```

This means we allow the constraint not to be satisfied. Therefore, it should not be posted.

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

CHAPTER 3

Solving problems

3.1 Finding solutions

Choco 3.2 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 Limits 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.

3.1.1 Satisfaction problems

Finding a solution

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

```
        // 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.lexico_LB(new IntVar[]{x, y}));
        // 5. Launch the resolution process
        solver.findSolution();
```

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.

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.

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).

3.1.2 Optimization problems

Choco 3.2 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);
```

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.

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.

3.1.3 Multi-objective optimization problems

Finding the pareto front

It is possible to solve a multi-objective optimization problems with Choco 3.2, 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:

```
a = VF.enumerated("a", 0, 2, solver);
b = VF.enumerated("b", 0, 2, solver);
c = VF.enumerated("c", 0, 2, solver);
solver.post(ICF.arithm(a, "+", b, "<", 3));
solver.findParetoFront(ResolutionPolicy.MAXIMIZE,a,b);
List<Solution> paretoFront = solver.getSolutionRecorder().getSolutions();
System.out.println("The pareto front has "+paretoFront.size()+" solutions : ");
for(Solution s:paretoFront){
        System.out.println("a = "+s.getIntVal(a)+" and b = "+s.getIntVal(b));
}
```

3.2 Recording solutions

Choco 3.2 requires each solution to be fully instantiated, i.e. every variable must be fixed. Otherwise, an exception will be thrown if assertions are turned on (when `-ea` is added to the JVM parameters). Choco 3.2 includes several ways to record solutions.

3.2.1 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.

3.2.2 Solution recording

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.

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:

```
                solver.plugMonitor(new IMonitorSolution() {
                        @Override
                        public void onSolution() {
                                bestObj = nbValues.getValue();
                                System.out.println("Solution found! Objective = "+bestObj);
                        }
                });
```

3.2.3 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.

3.3 Search Strategies

If no search strategy is specified in the model, Choco 3.2 will generate a default one. 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(...)`.

`IntStrategyFactory` offers several built-in search strategies and a simple framework to build custom searches.

3.4 Logging

Choco 3.2 has a simple logger which can be used by calling

```
SearchMonitorFactory.log(Solver solver, boolean solution, boolean choices);
```

The first argument is the solver. The second indicates whether or not each solution (and associated resolution statistics) should be printed. The third argument indicates whether or not each branching decision should be printed. This may be useful for debugging.

In general, in order to have a reasonable amount of information, we set the first boolean to true and the second to false.

3.5 Limiting the resolution

3.5.1 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.

3.5.2 Custom search limits

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

```
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.

CHAPTER 4

Advanced usage

4.1 Overview of Choco 3.2

4.1.1 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 5

Inside Choco

CHAPTER 6

Appendix

6.1 Graph constraints

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.

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