Choco3 Documentation

Release 3.2.0

Charles Prud'homme, Jean-Guillaume Fages

September 09, 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 What is Choco ?

Choco is a Free and Open-Source Software [1] dedicated to Constraint Programming. It aims at describing real combinatorial problems in the form of Constraint Satisfaction Problems and to solve them with Constraint Programming techniques.

Choco can be used for:

- teaching (a user-oriented constraint solver with open-source code)
- research (state-of-the-art algorithms and techniques, user-defined constraints, domains and variables)
- real-life applications (many application now embed CHOCO)

Choco is easy to manipulate, that's why it is widely used for teaching. And Choco is also efficient, and we are proud to count industrial users too.

Choco is developed with Intellij IDEA and JProfiler.

Choco is one of the few Java libraries for constraint programming. The first version dates from the early 2000s, Choco is one of the forerunners among the free solvers - Choco written under BSD license. Maintenance and development tools are provided by the members of INRIA TASC team, especially by Charles Prud'homme and Jean-Guillaume Fages [2]. The latest version is Choco 3.2.

Choco 3.2 is not the continuation of Choco2, but a completely rewritten version and there is no backward compatibility. The current release, choco-solver-3.2.0, is hosted on GitHub. Choco 3.2 comes with:

- various type of variables (integer, boolean, set, graph and real),
- various state-of-the-art constraints (alldifferent, count, nvalues, etc.),
- various search strategies, from basic ones (first_fail, smallest, etc.) 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, a FlatZinc parser, facilities to interact with the search loop, factories to help modeling, many samples, Choco-Ibex interface, etc.

An overview of the features of Choco 3.2 can be found in the presentation made in the "CP Solvers: Modeling, Applications, Integration, and Standardization" workshop of CP2013.

A forum is available on the website of Choco. A support mailing list is also available: choco3-support@mines-nantes.fr.

[1] Choco is distributed under BSD license (Copyright(c) 1999-2014, Ecole des Mines de Nantes).

[2] A complete list of contributors can be found on the website of Choco, team page.

1.2 By the way, 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]

1.3 Installing Choco 3.2

Choco 3.2 is a java library based on Java 7. 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 not include `choco-solver`.

1.3.1 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 An ready-to-use jar file ; 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-3.2.0-sources.jar The source of the core library.

choco-samples-3.2.0-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.

There are also official extensions, thus maintained by the Choco team. They are provided apart from the zip file. Each of the following extensions include dependencies but choco-solver classes, which ease their usage. The available extensions are:

choco-parsers-3.2.0.jar This extension provides tools to parse modelling languages to Choco; it should be selected to work with MiniZinc and FlatZinc files.

choco-gui-3.2.0.jar This extension provides a Graphical User Interface to interact and visualize the search process.

choco-cpviz-3.2.0.jar This extension produces files required for the cpviz software.

choco-geost-3.2.0.jar This extension provides support for the well-known Geost constraint, which is almost a solver by itself.

choco-exppar-3.2.0.jar This extension provides an expression parser to ease modelling.

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

To start using Choco 3.2, 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.

1.3.2 Update the classpath

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

```
java -cp .:choco-solver-3.2.0.jar my.project.Main
java -cp .:choco-solver-3.2.0.jar:choco-parsers-3.2.0.jar my.other.project.Main
```

1.3.3 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>
```

1.3.4 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
```

1.4 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 integer variables
`SetConstraintFactory`	SCF	Constraints over set variables
`GraphConstraintFactory`	GCF	Constraints over graph variables
`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:

```
import static 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 3.2 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.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.

1.5 Choco 3.2 quick documentation

1.5.1 Solver

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

[Solver] , *Limiting the resolution* .

1.5.2 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]

1.5.3 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]

1.5.4 Constants

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

[Constants]

1.5.5 Constraints

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

1.5.6 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 `GraphStrategyFactory`.

1.5.7 Large Neighborhood Search (LNS)

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

1.5.8 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.plugSearchMonitor(...)`.

1.5.9 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.

1.5.10 Restarts

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

1.5.11 Logging

Logging the search is possible. There are variants but the main way to do it is made through the `SMF.log(Solver, boolean, boolean)`. The first boolean indicates whether or not logging solutions, the second indicates whether or not logging search decisions. It also print, by default, main statistics of the search (time, nodes, fails, etc.)

1.5.12 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.

1.5.13 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`.

1.5.14 Explanations

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

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, 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.

2.1.1 Getters

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 and boolean 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 set (ie whose *KIND* is set to *REAL*)
`GraphVar[] retrieveGraphVars()`	Extract from the solver variables those which are graph (ie whose *KIND* is set to *GRAPH*)

Constraints

Method	Definition
`Constraint[] getCstrs()`	Return the array of constraints posted in the solver.
`getNbCstrs()`	Return the number of constraints posted in the solver.

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.
`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.

2.1.2 Setters

Method	Definition
`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 `TwoBucketPropagationEngine`.
`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*.

2.1.3 Others

Method	Definition
`void plugMonitor(ISearchMonitor sm)`	Put a *search monitor* to react on search events (solutions, decisions, fails, ...).
`Solver duplicateModel()`	Duplicate the model associates with a solver, ie only variables and constraints, and return a new solver.

2.2 Declaring variables

2.2.1 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 3.2 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.

2.2.2 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.

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);
```

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 type of constraints used. 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.

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);
```

2.2.3 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.

2.2.4 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 = 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);
```

2.2.5 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 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);
```

2.2.6 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.

2.2.7 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", 0.2d, 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);
```

2.3 Constraints and propagators

2.3.1 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 3.2 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:

```
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.

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 3.2 provides various types of constraints: *unary constraints*, *binary constraints*, *ternary constraints* and *global constraints*. A constraint should either be posted or be reified.

2.3.2 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

2.3.3 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:

```
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.

Reifying a constraint means that we allow the constraint not to be satisfied. Therefore, the constraint **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 *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.

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.

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).

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 $(X_0 < b_0 \lor X_1 < b_1 \lor \ldots \lor X_n < b_n$ 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));
}
```

3.1.4 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.

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:

```java
                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:

```java
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

3.3.1 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|release| 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.

3.3.2 Default 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(...)`. The default search strategy distincts variables per types and defines a specific search strategy per each type:

1. integer variables (but boolean variables: `IntStrategyFactory.minDom_LB(ivars)`
2. boolean variables: `IntStrategyFactory.lexico_UB(bvars)`
3. set variables: `SetStrategyFactory.force_minDelta_first(svars)`
4. graph variables `GraphStrategyFactory.graphLexico(gvar)`
5. real variables `new RealStrategy(rvars, new Cyclic(), new RealDomainMiddle())`

Constants are excluded from search strategies' variable scope.

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

3.3.3 A search strategy

It is strongly recommended to adapt the search space exploration to the problem treated. To do so, one can use built-in search strategies provided in `IntSearchStrategy`, `SetStrategyFactory` and `GraphStrategyFactory`.

It is also possible to create an assignment strategy (over integer variables) by using :

```
IntSearchStrategy.custom(VAR_SELECTOR, VAL_SELECTOR, VARS)
```

or:

```
IntSearchStrategy.custom(VAR_SELECTOR, VAL_SELECTOR, DEC_OPERATOR, VARS)
```

In the first case, the `DEC_OPERATOR` is set to `IntSearchStrategy.assign()`. Such methods required the declaraion of a :

1. `VAR_SELECTOR`: a variable selector, defines how to select the next variable to branch on,
2. `VAL_SELECTOR`: a value selector, defines how to select a value in the domain of the selected variable,
3. `DEC_OPERATOR`: a decision operator, defines how to modify the domain of the selected variable with the selected value,
4. `VARS`: sets of variables to branch on.

Some `VariableSelector`, `IntValueSelector` and `DecisionOperator` are provided in `IntSearchStrategy`.

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

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 Choco|release|.

3.3.4 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 $(k+1)^th$ prefix is the k^th prefix repeated `grow` times and immediately followed by coefficient $\texttt{grow}^k$.

- 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,...]

3.3.5 Limiting the resolution

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.

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

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.

Going further

Large Neighborhood Search, *Multi-threading*, *Explanations*.

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.

If the two booleans are set to false, the trace would start with a welcome message:

```
** Choco 3.2.0 (2014-05) : Constraint Programming Solver, Copyleft (c) 2010-2014
** Solve : myProblem
```

Then, when the resolution process ends, a complementary message is printed, based on the measures recorded.

```
- Search complete - [ No solution. ]
   Solutions: {0}
[  Maximize = {1}  ]
[  Minimize = {2}  ]
   Building time : {3}s
   Initialisation : {4}s
   Initial propagation : {5}s
   Resolution : {6}s
   Nodes: {7}
   Backtracks: {8}
   Fails: {9}
   Restarts: {10}
   Max depth: {11}
   Propagations: {12} + {13}
   Memory: {14}mb
   Variables: {15}
   Constraints: {16}
```

Brackets *[instruction]* indicate an optional instruction. If no solution has been found, the message “No solution.” appears on the first line. `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 *fine* propagations
13. number of *coarse* propagations
14. estimation of the memory used
15. number of variables in the model
16. 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 formated thanks to the `IMeasureRecorder` which is a *search monitor*.

When the first boolean of `SearchMonitorFactory.log(Solver, boolean, boolean);` is set to true, 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).

When the second boolean of `SearchMonitorFactory.log(Solver, boolean, boolean);` is set to true, on each node 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]`.

Warning: Printing the choices slows down the search process.

CHAPTER 4

Advanced usage

4.1 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.

4.1.1 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(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(100)`), a restart is forced. Then, every `30` calls to this neighborhood, the number of fixed is randomly picked. `20140909L` is the seed for the `java.util.Random`.

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

```
INeighbor neighbor = random(solver, vars, level, seed, frcounter);
LargeNeighborhoodSearch lns = new LargeNeighborhoodSearch(solver, neighbor, true);
solver.getSearchLoop().plugSearchMonitor(lns);
```

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.

4.1.2 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.

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.

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`$\leq$ 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.

Defining its own neighborhoods

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

Method	Definition
`void recordSolution()`	Action to perform on a solution (typicallu, storing the current variables' value).
`void fixSomeVariables(ICause cause) throws ContradictionException`	Fix some variables to their value in the last solution, computing a partial solution.
`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.

4.1.3 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].

4.1.4 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.

4.2 Multi-threading

4.3 Explanations

4.4 Search monitor

4.4.1 Principle

4.5 Defining its own search strategy

4.6 Defining its own constraint

4.7 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

[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.

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.

4.8 Ibex

> "IBEX is a C++ library for constraint processing over real numbers.
>
> It provides reliable algorithms for handling non-linear constraints. In particular, roundoff 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.

4.8.1 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=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.

CHAPTER 5

Inside Choco

CHAPTER 6

Appendix

6.1 Available constraints in Choco 3.2.0

6.1.1 Unary constraints

Constraint involving only one variable.

6.1.2 Binary constraints

Constraint involving only two variables.

6.1.3 Ternary constraints

Constraint involving only three variables.

6.1.4 Global constraints

Constraint involving an unfixed number of variables.

6.1.5 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*