CHOCO Documentation

http://choco.emn.fr/

October 16, 2009

Contents

Preface

Choco is a java library for constraint satisfaction problems (CSP), constraint programming (CP) and explanation-based constraint solving (e-CP). It is built on a event-based propagation mechanism with backtrackable structures. Choco is an open-source software, distributed under a **BSD licence** and hosted by sourceforge.net. For any requests send a mail to choco@emn.fr.

This document is organized as follows:

- Documentation is the user-guide of Choco. After a short introduction to constraint programming and to the Choco solver, it presents the basics of modeling and solving with Choco, the advanced usages (customizing propagation and search), some examples of Applications, and a FAQ section.
- Elements of Choco gives a detailed description of the variables, operators, constraints currently available in Choco.
- Tutorials provides a fast how-to write a Choco program, a detailed example of a simple program, and several exercises with their solutions.
- Extras presents future works, only available on the beta version or extension of the current jar, such as the visualization module of Choco. The section dedicated to Sudoku aims at explaining the basic principles of Constraint Programming (propagation and search) on this famous game.

Part I

Documentation

The documentation of Choco is organized as follows:

- The concise introduction provides some informations about constraint programming concepts and a "Hello world"-like first Choco program.
- The model section gives informations on how to create a model and introduces variables and constraints.
- The solver section gives informations on how to create a solver, to read a model, to define a search strategy, and finally to solve a problem.
- The advanced use section explains how to define your own limit search space, search strategy, constraint, operator, variable, or backtrackable structure.
- The applications section shows the use of Choco defined global constraints on scheduling or placement problems.

Chapter 1

Introduction to constraint programming and Choco

1.1 About constraint programming

> Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.
>
> **E. C. Freuder, Constraints, 1997.**

Fast increasing computing power in the 1960s led to a wealth of works around problem solving, at the root of Operational Research, Numerical Analysis, Symbolic Computing, Scientific Computing, and a large part of Artificial Intelligence and programming languages. Constraint Programming is a discipline that gathers, interbreeds, and unifies ideas shared by all these domains to tackle decision support problems.

Constraint programming has been successfully applied in numerous domains. Recent applications include computer graphics (to express geometric coherence in the case of scene analysis), natural language processing (construction of efficient parsers), database systems (to ensure and/or restore consistency of the data), operations research problems (scheduling, routing), molecular biology (DNA sequencing), business applications (option trading), electrical engineering (to locate faults), circuit design (to compute layouts), etc.

Current research in this area deals with various fundamental issues, with implementation aspects and with new applications of constraint programming.

1.1.1 Constraints

A constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain. A constraint thus restricts the possible values that variables can take, it represents some partial information about the variables of interest. For instance, the circle is inside the square relates two objects without precisely specifying their positions, i.e., their coordinates. Now, one may move the square or the circle and he or she is still able to maintain the relation between these two objects. Also, one may want to add another object, say a triangle, and to introduce another constraint, say the square is to the left of the triangle. From the user (human) point of view, everything remains absolutely transparent.

Constraints naturally meet several interesting properties:

- constraints may specify partial information, i.e. constraint need not uniquely specify the values of its variables,

- constraints are non-directional, typically a constraint on (say) two variables X, Y can be used to infer a constraint on X given a constraint on Y and vice versa,
- constraints are declarative, i.e. they specify what relationship must hold without specifying a computational procedure to enforce that relationship,
- constraints are additive, i.e. the order of imposition of constraints does not matter, all that matters at the end is that the conjunction of constraints is in effect,
- constraints are rarely independent, typically constraints in the constraint store share variables.

Constraints arise naturally in most areas of human endeavor. The three angles of a triangle sum to 180 degrees, the sum of the currents floating into a node must equal zero, the position of the scroller in the window scrollbar must reflect the visible part of the underlying document, these are some examples of constraints which appear in the real world. Thus, constraints are a natural medium for people to express problems in many fields.

1.1.2 Constraint Programming

Constraint programming is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints (conditions, properties) which must be satisfied by the solution.

Work in this area can be tracked back to research in Artificial Intelligence and Computer Graphics in the sixties and seventies. Only in the last decade, however, has there emerged a growing realization that these ideas provide the basis for a powerful approach to programming, modeling and problem solving and that different efforts to exploit these ideas can be unified under a common conceptual and practical framework, constraint programming.

> If you know **sudoku**, then you know **constraint programming**. See why here.

1.2 Modeling with Constraint programming

The formulation and the resolution of combinatorial problems are the two main goals of the constraint programming domain. This is an essential way to solve many interesting industrial problems such as scheduling, planning or design of timetables. The main interest of constraint programming is to propose to the user to model a problem without being interested in the way the problem is solved.

1.2.1 The Constraint Satisfaction Problem

Constraint programming allows to solve combinatorial problems modeled by a Constraint Satisfaction Problem (CSP). Formally, a CSP is defined by a triplet (X, D, C):

- **Variables**: $X = \{X_1, X_2, \ldots, X_n\}$ is the set of variables of the problem.
- **Domains**: D is a function which associates to each variable X_i its domain $D(X_i)$, i.e. the set of possible values that can be assigned to X_i. The domain of a variable is usually a finite set of integers: $D(X_i) \subset \mathbb{Z}$ (*integer variable*). But a domain can also be continuous ($D(X_i) \subseteq \mathbb{R}$ for a *real variable*) or made of discrete set values ($D(X_i) \subseteq \mathcal{P}(\mathbb{Z})$ for a *set variable*).
- **Constraints**: $C = \{C_1, C_2, \ldots, C_m\}$ is the set of constraints. A constraint C_j is a relation defined on a subset $X^j = \{X_1^j, X_2^j, \ldots, X_{n^j}^j\} \subseteq X$ of variables which restricts the possible tuples of values $(v_1, \ldots, v_{n^j})$ for these variables:

  $$(v_1, \ldots, v_{n^j}) \in C_j \cap (D(X_1^j) \times D(X_2^j) \times \cdots \times D(X_{n^j}^j)).$$

  Such a relation can be defined explicitely (ex: $(X_1, X_2) \in \{(0,1),(1,0)\}$) or implicitely (ex: $X_1 + X_2 \leq 1$).

Solving a CSP is to find a tuple $v = (v_1, \ldots, v_n) \in D(X)$ on the set of variables which satisfies all the constraints:

$$(v_1, \ldots, v_{n^j}) \in C_j, \quad \forall j \in \{1, \ldots, m\}.$$

For optimization problems, one need to define an **objective function** $f : D(X) \rightarrow \mathbb{R}$. An optimal solution is then a solution tuple of the CSP that minimizes (or maximizes) function f.

1.2.2 Examples of CSP models

This part provides three examples using different types of variables in different problems. These examples are used throughout this tutorial to illustrate their modeling with Choco.

Example 1: the n-queens problem.

Let us consider a chess board with n rows and n columns. A queen can move as far as she pleases, horizontally, vertically, or diagonally. The standard n-queens problem asks how to place n queens on an n-ary chess board so that none of them can hit any other in one move.

The n-queens problem can be modeled by a CSP in the following way:

- **Variables**: $X = \{X_i \mid i \in [1, n]\}$.
- **Domain**: for all variable $X_i \in X$, $D(X_i) = \{1, 2, \ldots, n\}$.
- **Constraints**: the set of constraints is defined by the union of the three following constraints,
  - queens have to be on distinct lines:
    - $C_{lines} = \{X_i \neq X_j \mid i, j \in [1, n], i \neq j\}$.
  - queens have to be on distinct diagonals:
    - $C_{diag1} = \{X_i \neq X_{j+j-i} \mid i, j \in [1, n], i \neq j\}$.
    - $C_{diag2} = \{X_i \neq X_{j+i-j} \mid i, j \in [1, n], i \neq j\}$.

Example 2: the ternary Steiner problem.

A ternary Steiner system of order n is a set of $n*(n-1)/6$ triplets of distinct elements taking their values in $[1, n]$, such that all the pairs included in two distinct triplets are different. See http://mathworld.wolfram.com/SteinerTripleSystem.html for details.

The ternary Steiner problem can be modeled by a CSP in the following way:

- let $t = n * (n - 1)/6$.
- **Variables**: $X = \{X_i \mid i \in [1, t]\}$.
- **Domain**: for all $i \in [1, t]$, $D(X_i) = \{1, ..., n\}$.
- **Constraints**:
  - every set variable X_i has a cardinality of 3:
    - for all $i \in [1, t]$, $|X_i| = 3$.
  - the cardinality of the intersection of every two distinct sets must not exceed 1:
    - for all $i, j \in [1, t]$, $i \neq j$, $|X_i \cap X_j| \leq 1$.

Example 3: the CycloHexane problem.

The problem consists in finding the 3D configuration of a cyclohexane molecule. It is described with a system of three non linear equations:

- **Variables**: x, y, z.
- **Domain**: $]-\infty;+\infty[$.
- **Constraints**:

$$y^2 * (1 + z^2) + z * (z - 24 * y) = -13$$
$$x^2 * (1 + y^2) + y * (y - 24 * x) = -13$$
$$z^2 * (1 + x^2) + x * (x - 24 * z) = -13$$

1.3 My first Choco program: the magic square

1.3.1 The magic square problem

In the following, we will address the magic square problem of order 3 to illustrate step-by-step how to model and solve this problem using choco.

Definition:

A magic square of order n is an arrangement of n^2 numbers, usually distinct integers, in a square, such that the n numbers in all rows, all columns, and both diagonals sum to the same constant. A standard magic square contains the integers from 1 to n^2.

The constant sum in every row, column and diagonal is called the magic constant or magic sum M. The magic constant of a classic magic square depends only on n and has the value: $M(n) = n(n^2+1)/2$.

More details on the magic square problem.

1.3.2 A mathematical model

Let x_{ij} be the variable indicating the value of the j^{th} cell of row i. Let C be the set of constraints modeling the magic square as:

$$x_{ij} \in [1, n^2], \qquad \forall i, j \in [1, n]$$
$$x_{ij} \neq x_{kl}, \qquad \forall i, j, k, l \in [1, n], i \neq k, j \neq l$$
$$\sum_{j=1}^{n} x_{ij} = n^2, \qquad \forall i \in [1, n]$$
$$\sum_{i=1}^{n} x_{ij} = n^2, \qquad \forall j \in [1, n]$$
$$\sum_{i=1}^{n} x_{ii} = n^2$$
$$\sum_{i=n}^{1} x_{i(n-i)} = n^2$$

We have all the required information to model the problem with Choco.

> For the moment, we just talk about *model translation* from a mathematical representation to Choco. Choco can be used as a *black box*, that means we just need to define the problem without knowing the way it will be solved. We can therefore focus on the modeling not on the solving.

1.3.3 To Choco...

First, we need to know some of the basic Choco objects:

- The **model** (object `Model` in Choco) is one of the central elements of a Choco program. Variables and constraints are associated to it.
- The **variables** (objects `IntegerVariable`, `SetVariable`, and `RealVariable` in Choco) are the *unknown* of the problem. Values of variables are taken from a **domain** which is defined by a set of values or quite often simply by a lower bound and an upper bound of the allowed values. The domain is given when creating the variable.

> Do not forget that we manipulate **variables** in the mathematical sense (as opposed to classical computer science). Their effective value will be known only once the problem has been solved.

- The **constraints** define relations to be satisfied between variables and constants. In our first model, we only use the following constraints provided by Choco:
  - `eq(var1, var2)` which ensures that `var1` equals `var2`.
  - `neq(var1, var2)` which ensures that `var1` is not equal to `var2`.
  - `sum(var[])` which returns expression `var[0]+var[1]+...+var[n]`.

1.3.4 The program

After having created your java class file, import the Choco class to use the API:

```
import static choco.Choco.*;
```

First of all, let's create a Model:

```
// Constant declaration
int n = 3; // Order of the magic square
int magicSum = n * (n * n +1) /2; // Magic sum

// Build the model
Model m = new CPModel();
```

We create an instance of `CPModel()` for **C**onstraint **P**rogramming Model. Do not forget to add the following imports:

```
import choco.cp.model.CPModel;
import choco.kernel.model.Model;
```

Then we declare the variables of the problem:

```
// Creation of an array of variables
IntegerVariable[][] var = new IntegerVariable[n][n];

// For each variable, we define its name and the boundaries of its domain.
for (int i = 0; i< n; i++){
   for (int j = 0; j< n; j++){
        var[i][j] = Choco.makeIntVar("var_" + i + "_" + j, 1, n * n);
        // Associate the variable to the model.
        m.addVariable(var[i][j]);
   }
}
```

Add the import:

```
import choco.kernel.model.variables.integer.IntegerVariable;
```

We have defined the variable using the `makeIntVar` method which creates an enumerated domain: all the values are stored in the java object (beware, it is usually not necessary to store all the values and it is less efficient than to create a bounded domain).
Now, we are going to state a constraint ensuring that all variables must have a different value:

```
// All cells of the matrix must be different
for (int i = 0; i < n*n; i++){
    for (int j = i + 1; j < n*n; j++){
       Constraint c = (Choco.neq(var[i / n][i % n],var[j / n][j % n]));
       m.addConstraint(c);
    }
}
```

Add the import:

```
import choco.kernel.model.constraints.Constraint;
```

Then, we add the constraint ensuring that the magic sum is respected:

```
// All rows must be equal to the magic sum
for(int i = 0; i < n; i ++){
   m.addConstraint(eq(sum(var[i]), magicSum));
}
```

Then we define the constraint ensuring that each column is equal to the magic sum. Actually, `var` just denotes the rows of the square. So we have to declare a temporary array of variables that defines the columns.

```
IntegerVariable[][] varCol = new IntegerVariable[n][n];
for(int i = 0; i < n; i ++){
   for(int j = 0; j < n; j ++){
        // Copy of var in the column order
        varCol[i][j] = var[j][i];
   }
   // Each column's sum is equal to the magic sum
   m.addConstraint(eq(sum(varCol[i]), magicSum));
}
```

It is sometimes useful to define some temporary variables to keep the model simple or to reorder array of variables. That is why we also define two other temporary arrays for diagonals.

```
IntegerVariable[] varDiag1 = new IntegerVariable[n];
IntegerVariable[] varDiag2 = new IntegerVariable[n];
for(int i = 0; i < n; i ++){
   varDiag1[i] = var[i][i]; // Copy of var in varDiag1
   varDiag2[i] = var[(n-1)-i][i]; // Copy of var in varDiag2
}
// Every diagonal's sum has to be equal to the magic sum
m.addConstraint(eq(sum(varDiag1), magicSum));
m.addConstraint(eq(sum(varDiag2), magicSum));
```

Now, we have defined the model. The next step is to solve it. For that, we build a Solver object:

```
// Build the solver
Solver s = new CPSolver();
```

with the imports:

```
import choco.kernel.solver.Solver;
import choco.cp.solver.CPSolver;
```

We create an instance of `CPSolver()` for Constraint Programming Solver. Then, the solver reads (translates) the model and solves it:

```
// Read the model
s.read(m);
// Solve the model
s.solve();
// Print the solution
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        System.out.print(s.getVar(var[i][j]).getVal()+" ");
   }
   System.out.println("");
}
```

The only variables that need to be printed are the ones in `var` (all the others are only references to these ones).

> We have to use the Solver to get the value of each variable of the model. The Model only declares the objects, the Solver finds their value.

We are done, we have created our first Choco program. The complete source code can be found here: ExMagicSquare.zip

1.3.5 In summary

- A Choco Model is defined by a set of Variables with a given domain and a set of Constraints that link Variables: it is necessary to add both Variables and Constraints to the Model.
- temporary Variables are useful to keep the Model readable, or necessary when reordering arrays.
- The value of a Variable can be known only once the Solver has found a solution.
- To keep the code readable, you can avoid the calls to the static methods of the Choco classes, by importing the static classes, i.e. instead of:

```
import choco.Choco;
...
IntegerVariable v = Choco.makeIntVar("v", 1, 10);
...
Constraint c = Choco.eq(v, 5);
```

you can use:

```
import static choco.Choco.*;
...
IntegerVariable v = makeIntVar("v", 1, 10);
...
Constraint c = eq(v, 5);
```

1.4 Complete examples

We provide now the complete Choco model for the three examples previously described.

1.4.1 Example 1: the n-queens problem with Choco

This first model for the n-queens problem only involves binary constraints of differences between integer variables. One can immediately recognize the 4 main elements of any Choco code. First of all, create

the model object. Then create the variables by using the Choco API (One variable per queen giving the row (or the column) where the queen will be placed). Finally, add the constraints and solve the problem.

```
///1- Create the problem
       Model m = new CPModel();

//2- Create the variables
       IntegerVariable[] queens = makeIntVarArray("Q", n, 1, n);

//3- Post constraints
       for (int i = 0; i < n; i++) {
           for (int j = i + 1; j < n; j++) {
               int k = j - i;
               m.addConstraint(neq(queens[i], queens[j]));
               m.addConstraint(neq(queens[i], plus(queens[j], k))); // diagonal constraints
               m.addConstraint(neq(queens[i], minus(queens[j], k))); // diagonal constraints
           }
       }

//4- Search for all solutions
       Solver s = new CPSolver();
       s.read(m);
       s.solveAll();

//5- Print the number of solution found
       System.out.println("NbSol:␣" + s.getNbSolutions());
```

1.4.2 Example 2: the ternary Steiner problem with Choco

The ternary Steiner problem is entirely modeled using set variables and set constraints.

```
//1- Create the problem
       Model mod = new CPModel();
       int m = 7;
       int n = m * (m - 1) / 6;

//2- Create Variables
       SetVariable[] vars = new SetVariable[n]; // A variable for each set
       SetVariable[] intersect = new SetVariable[n * n]; // A variable for each pair of sets
       for (int i = 0; i < n; i++)
           vars[i] = makeSetVar("set␣" + i, 1, n);
       for (int i = 0; i < n; i++)
           for (int j = i + 1; j < n; j++)
               intersect[i * n + j] = makeSetVar("interSet␣" + i + "␣" + j, 1, n);

//3- Post constraints
       for (int i = 0; i < n; i++)
           mod.addConstraint(eqCard(vars[i], 3));
       for (int i = 0; i < n; i++)
           for (int j = i + 1; j < n; j++) {
               // the cardinality of the intersection of each pair is equal to one
               mod.addConstraint(setInter(vars[i], vars[j], intersect[i * n + j]));
               mod.addConstraint(leqCard(intersect[i * n + j], 1));
           }

//4- Search for a solution
       Solver s = new CPSolver();
       s.read(mod);
       s.setVarSetSelector(new MinDomSet(s, s.getVar(vars)));
       s.setValSetSelector(new MinEnv(s));
       s.solve();
```

```
//5- Print the solution
       for (int i = 0; i < n; i++)
           System.out.println(s.getVar(vars[i]).pretty());
```

1.4.3 Example 3: the CycloHexane problem with Choco

Real variables are illustrated on the problem of finding the 3D configuration of a cyclohexane molecule.
WARNING: This example doesn't work anymore!!

```
//1- Create the problem
       Model pb = new CPModel();
       pb.setPrecision(1e-8);

//2- Create the variable
       RealVariable x = makeRealVar("x", -1.0e8, 1.0e8);
       RealVariable y = makeRealVar("y", -1.0e8, 1.0e8);
       RealVariable z = makeRealVar("z", -1.0e8, 1.0e8);

//3- Create and post the constraints
       RealExpressionVariable exp1 = plus(mult(power(y, 2), plus(1, power(z, 2))),
              mult(z, minus(z, mult(24, y))));

       RealExpressionVariable exp2 = plus(mult(power(z, 2), plus(1, power(x, 2))),
              mult(x, minus(x, mult(24, z))));

       RealExpressionVariable exp3 = plus(mult(power(x, 2), plus(1, power(y, 2))),
              mult(y, minus(y, mult(24, x))));

       Constraint eq1 = eq(exp1, -13);
       Constraint eq2 = eq(exp2, -13);
       Constraint eq3 = eq(exp3, -13);

       pb.addConstraint(eq1);
       pb.addConstraint(eq2);
       pb.addConstraint(eq3);

//4- Search for all solution
       Solver s = new CPSolver();
       s.read(pb);
       s.setVarRealSelector(new CyclicRealVarSelector(s));
       s.setValRealIterator(new RealIncreasingDomain());

//5- print the solution found
       System.out.println("x␣" + s.getVar(x).getValue());
       System.out.println("y␣" + s.getVar(y).getValue());
       System.out.println("z␣" + s.getVar(z).getValue());
```

Chapter 2

The model

The Model, along with the Solver, is one of the two key elements of any Choco program. The Choco Model allows to describe a problem in an easy and declarative way: it simply records the variables and the constraints defining the problem.

This section describes the large API provided by Choco to create different types of variables and constraints.

Note that a static import is required to use the Choco API:

```
import static choco.Choco.*;
```

First of all, a Model object is created as follows:

```
Model model = new CPModel();
```

In that specific case, a Constraint Programming (CP) Model object has been created.

2.1 Variables

A Variable is defined by a type (integer, real, or set variable), a name, and the values of its domain. When creating a simple variable, some options can be set to specify its domain representation (ex: enumerated or bounded) within the Solver.

> The choice of the domain should be considered. The efficiency of the solver often depends on judicious choice of the domain type.

Variables can be combined as expression variables using operators.

One or more variables can be added to the model using the following methods of the `Model` class:

```
model.addVariable(Variable... var);
model.addVariable(String options, Variable... var);
```

Example

adding two variables to the model

```
model.addVariable(var1, var2);
```

options allows to define the specific role of variables *var*: decision/non-decision variables or objective variable.

2.1.1 Simple Variables

See Section Variables for details:

IntegerVariable, SetVariable, RealVariable

2.1.2 Expression variables and operators

Expression variables represent the result of combinations between variables of the same type made by operators. Three types of expression variables exist :

`IntegerExpressionVariable`, `SetExpressionVariable`, and `RealExpressionVariable`.

One can define a buffered expression variable to make a constraint easy to read, for example:

```
IntegerVariable v1 = makeIntVar("v1", 1, 3);
IntegerVariable v2 = makeIntVar("v2", 1, 3);
IntegerExpressionVariable v1Andv2 = plus(v1, v2);
```

To construct expressions of variables, simple operators can be used. Each returns a `ExpressionVariable` object:

abs, cos, div, FALSE, max, min, minus, mod, mult, neg, plus, powerscalar, sin, sum, TRUE.

Note that these operators are not considered as constraints: they do not return a `Constraint` objet but a `Variable` object.

2.1.3 Constant variables

Under development

2.1.4 Decision/non-decision variables

Once all the variables of the problem has been declared, it is possible to specify which variables are decision variables or non-decision variables. It can be done when adding the variables to the Model, by setting the option to `cp:decision` or `cp:no_decision`.

```
IntegerVariable toto = Choco.makeIntVar("toto", 1, 2, "cp:decision"); // decision variable
IntegerVariable titi = Choco.makeIntVar("titi", 1, 2, "cp:no_decision"); // non-decision
```

Each of these options can also be set within a single instruction for a group of variables, as follows:

```
IntegerVariable toto = Choco.makeIntVar("toto", 1, 2);
IntegerVariable titi = Choco.makeIntVar("titi", 1, 2);
model.addVariable("cp:decision",toto, titi);
```

These options are useful when:

- one knows which variables are decision or non-decision ones
- one does not want some variables to be in the search strategy (then set `cp:no_decision`)
- one does not want to deal with a specialized search strategy

A `default` search strategy will be created on the decision variables.

- The declaration of a user-defined search strategy will erase setting `cp:decision`.
- The declaration of a search strategy will erase setting `cp:no_decision`.

more precise: user-defined/pre-defined, variable and/or value heuristics ?

2.1.5 Objective variable

You can define an objective variable directly within the model, by using option `cp:objective`:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 1000, "cp:objective");
IntegerVariable y = makeIntVar("y", 20, 50);
m.addConstraint(eq(x, mult(y, 20)));
s.read(m);
s.minimize(true);
```

Only one variable can be defined as an objective. If more than one objective variable is declared, then only the last one will be taken into account.

Note that optimization problems can be declared without defining an objective variable within the model (see the optimization example.)

2.1.6 Examples with variables

```
Model m = new CPModel();

// Bounded integer variables
IntegerVariable biv = makeIntVar("biv", 1, 10000, "cp:bound");
IntegerVariable[][] biVars = makeIntVarArray("biVars", 20, 30, 1, 99999, "cp:bound");
m.addVariable(biv);
for(int i = 0; i < biVars.length; i++){
  m.addVariable(biVars[i]);
}

// Enumerated integer variables
IntegerVariable eiv = makeIntVar("eiv", 1, 100, "cp:enum");
IntegerVariable[] eiVars = makeIntVarArray("eiVars", 10, 1, 50, "cp:enum");
int[] values1 = new int[]{1,3,5,7,14,16,18,24,46,78, 99};
IntegerVariable eiv2 = makeIntVar("eiv2", values,"cp:enum");
m.addVariable(eiv, eiv2);
m.addVariable(eiVars);

// Linked list integer variables
IntegeVariable lliv = makeIntVar("lliv", 1, 10, "cp:link");
int[] values2 = new int[]{1,3,5,7,14,16,18,24,46,78, 99};
IntegeVariable lliv2 = makeIntVar("lliv2", values2, "cp:link");
m.addVariable(lliv, lliv2);

IntegerExpressionVariable iev = plus(lliv, 1);
m.addVariable(iev);

// Bounded set variable
SetVariable bsv = makeSetVar("bsv", 1, 24, "cp:enum");
SetVariable bsv2 = makeSetVar("bsv2", 1, 36, "cp:bound");
```

```
// Enumerated set variable
SetVariable esv = makeSetVar("bsv", 1, 240,"cp:enum");

m.addVariable(bsv, bs2, esv);

// Real variable
RealVariable rv = makeRealVar("rv", -1.0, 1.0);
RealVariable rv2 = makeRealVar("rv", -8.77, 9.87);
RealExpressionVariable rev = plus(cos(rv), power(rv2, 2));
m.addVariable(rv, rv2, rev);
```

2.2 Constraints

Choco provides a large number of simple and global constraints and allows the user to easily define its own new constraint. A constraint deals with one or more variables of the model and specify conditions to be held on these variables. A constraint is stated into the model by using the following methods available from the `Model` API:

```
addConstraint(Constraint... c);
addConstraint(String options, Constraint... c);
```

Example

: adding a difference (disequality) constraint between two variables of the model

```
model.addConstraint(neq(var1, var2));
```

Available *options* depend on the kind of constraint c to add: they allow, for example, to choose the filtering algorithm to run during propagation. These are strings prefixed with `cp:`, such as `"cp:decomp"` or `"cp:ac"`.

This section presents the constraints available in the Choco API sorted by type or by domain. Related sections:

- a detailed description (with options, examples, references) of each constraint is given in Section constraints
- Section applications shows how to apply some specific global constraints
- Section user-defined constraint explains how to create its own constraint.

2.2.1 Binary constraints

Constraints involving two integer variables

- `eq`, `geq`, `gt`, `leq`, `lt`, `neq`
- `abs`, `oppositeSign`, `sameSign`

2.2.2 Ternary constraints

Constraints involving three integer variables

- `distanceEQ`, `distanceNEQ`, `distanceGT`, `distanceLT`
- `intDiv`, `mod`, `times`

2.2.3 Constraints involving real variables

Constraints involving two real variables

- eq, geq, leq

2.2.4 Constraints involving set variables

Set constraints are illustrated on the ternary Steiner problem.

- eqCard, geqCard, leqCard
- member, notMember
- isIncluded, isNotIncluded, setDisjoint
- setInter, setUnion
- max, min
- pack

2.2.5 Channeling constraints

The use of a redundant model is a frequent technique to strengthen propagation or to get more freedom to design dedicated search heuristics. The following constraints allow to ensure integrity of different models:

- inverseChanneling, boolChanneling

More complex channeling can be done using reified constraints (see Section reification) although they are less efficient. For example, to ensure that two variables are equal or not, one can reify the equality into a boolean variables :

```
IntegerVariable reifiedB = makeIntVar("bvar", 0, 1);
IntegerVariable x = makeIntVar("a", 0, 10);
IntegerVariable y = makeIntVar("b", 0, 10);
model.addConstraint(reifiedIntConstraint(reifiedB, eq(x, y)));
```

2.2.6 Constraints in extension and relations

Choco supports the statement of constraints defining arbitrary relations over two or more variables. Such a relation may be defined by three means:

- **feasible table:** the list of allowed tuples of values (that belong to the relation),
- **infeasible table:** the list of forbidden tuples of values (that do not belong to the relation),
- **predicate:** a method to be called in order to check whether a tuple of values belongs or not to the relation.

On the one hand, constraints based on tables may be rather memory consuming in case of large domains, although one relation table may be shared by several constraints. On the other hand, predicate constraints require little memory as they do not cache truth values, but imply some run-time overhead for calling the feasibility test. Table constraints are thus well suited for constraints over small domains; while predicate constraints are well suited for situations with large domains.

Different levels of consistency can be enforce on constraints in extension:

- several arc-consistency (AC) algorithms for binary relations
- two AC algorithms for n-ary relations dedicated either to positive or to negative tables (relation defined by the allowed or forbidden tuples)
- a weaker forward-checking (FC) algorithm for n-ary relations.

The Choco API for creating constraints in extension are as follows:

- feasPairAC, infeasPairAC, relationPairAC
- feasTupleAC, infeasTupleAC, relationTupleAC
- feasTupleFC, infeasTupleFC, relationTupleFC

Relations.

A same relation might be shared among several constraints, in this case it is highly recommended to create it first and then use the relationPairAC, relationTupleAC, or relationTupleFC API on the same relation for each constraint.

For binary relations, the following Choco API is provided:

```
makeBinRelation(int[] min, int[] max, List<int[]>pairs, boolean feas);
```

It returns a `BinRelation` giving a list of compatible (`feas=true`) or incompatible (`feas=false`) pairs of values. This relation can be applied to any pair of variables (x_1, x_2) whose domains are included in the `min/max` intervals, i.e. such that:

$$\texttt{min}[i] \leq x_i.\texttt{getInf}() \leq x_i.\texttt{getSup}() \leq \texttt{max}[i], \quad \forall i.$$

Bounds `min/max` are mandatory in order to allow to compute the opposite of the relation if needed.

For n-ary relations, the corresponding Choco API is:

```
makeLargeRelation(int[] min, int[] max, List<int[]> tuples, boolean feas);
```

It returns a `LargeRelation`. If `feas=true`, the returned relation matches also the `IterLargeRelation` interface which provides constant time iteration abilities over tuples (for compatibility with the GAC algorithm used over feasible tuples).

```
LargeRelation r = makeLargeRelation(min, max, tuples, true);
model.addConstraint(relationTupleAC(vars, r));
```

Lastly, some specific relations can be defined without storing the tuples, as in the following example (`TuplesTest` extends `LargeRelation`):

```
public class NotAllEqual extends TuplesTest {
    public boolean checkTuple(int[] tuple) {
        for (int i = 1; i < tuple.length; i++) {
            if (tuple[i - 1] != tuple[i]) return true;
        }
        return false;
    }
}
```

Then, a *NotAllEqual* constraint can be stated within the problem by:

```
model.addConstraint(relationTupleFC(new IntegerVariable[]{x, y, z}, new NotAllEqual()));
```

2.2.7 Reified constraints

Constraints involved in another constraint are usually called reified constraints. Typical examples of reified constraints are constraints combined with logical operators, such as $(x \neq y) \vee (z \leq 9)$.

Choco provides a generic constraint to reify any constraints on integer variables into a boolean variable expressing its truth value:

- `reifiedIntConstraint`

This mechanism can be used for example to model MaxCSP problems where the number of satisfied constraints has to be maximized. It is also intended to give the freedom to the user to build complex reified constraints. However, Choco provides a more simple and direct API to build complex expressions using boolean operators:

- `and`, `or`, `implies`, `ifOnlyIf`, `ifThenElse`, `not`

Such an expression is represented as a tree of operators. The leaves of this tree are made of variables, constants or even traditional constraints. Variables and constants can be combined as `ExpressionVariable` using operators (e.g, `mult(10,abs(w))`), or using simple constraints (e.g., `leq(z,9)`), or even using global constraints (e.g, `alldifferent(vars)`). The language available on expressions is therefore slightly richer and matches the language used in the Constraint Solver Competition 2008 of the CPAI workshop.

For example, the following expression

$$((x = 10 * |y|) \vee (z \leq 9)) \quad \Longleftrightarrow \quad \texttt{alldifferent}(a, b, c)$$

could be represented by :

```
Constraint exp = ifOnlyIf( or( eq(x, mult(10, abs(y))), leq(z, 9) ),
                           alldifferent(new IntegerVariable[]{a,b,c}) );
```

Handling complex expressions.

Expressions offer a more powerful modeling language than the one available via standard constraints. However, they can not be handled as efficiently as the standard constraints that embed a dedicated propagation algorithm. We therefore recommend you to carefully check that you can not model the expression using the intensional constraints of Choco before using expressions. Inside the solver, expressions can be represented in two different ways that can be decided at the modeling level, using the following Model API:

```
setDefaultExpressionDecomposition(boolean decomp);
```

- The first way (`decomp=false`) is to handle them as constraints in extension. The expression is then used to check a tuple in a dynamic way just like a n-ary relation that is defined without listing all the possible tuples. The expression is then propagated using the GAC3rm algorithm. This is very powerful as arc-consistency is obtained on the corresponding constraints.
- The second way (`decomp=true`) is to decompose the expression automatically by introducing intermediate variables and eventually the generic `reifiedIntConstraint`. By doing so, the level of pruning decreases but expressions of larger arity involving large domains can be represented.

Once the default representation is chosen, one can also make exception for a particular expression using options on `addConstraint`. For example, the following code tells the solver to decompose e1 and not e2 :

```
model.setDefaultExpressionDecomposition(false);
IntegerVariable x = makeIntVar("x", 1, 3, "cp:bound");
IntegerVariable y = makeIntVar("y", 1, 3, "cp:bound");
IntegerVariable z = makeIntVar("z", 1, 3, "cp:bound");

Constraint e1 = or(lt(x, y), lt(y, x));
model.addConstraint("cp:decomp", e1);

Constraint e2 = or(lt(y, z), lt(z, y));
model.addConstraint(e2);
```

When and how should I use expressions ?

An expression (represented in extension) should be used in the case of a complex logical relationship that involves **few different variables**, each of **small domain**, and if **arc consistency** is desired on those variables. In such a case, an expression can even be more powerful than a model using intermediate variables and intensional constraints. Imagine the following "crazy" example :

```
or( and( eq( abs(sub(div(x,50),div(y,50))),1), eq( abs(sub(mod(x,50),mod(y,50))),2)),
    and( eq( abs(sub(div(x,50),div(y,50))),2), eq( abs(sub(mod(x,50),mod(y,50))),1)))
```

This expression has a small arity: it involves only two variables x and y. Let assume that their domains has no more than 300 values, then such an expression should typically not be decomposed. Indeed, arc consistency will create many holes in the domains and filter much more than if the relation was decomposed.

Conversely, an expression should be decomposed as soon as it involves a large number of variables, or at least one variable with a large domain.

2.2.8 Global constraints

Choco includes several global constraints, such as:

```
allDifferent, globalCardinality, atMostNValue, cumulative, lex, regular, tree,
geost, etc.
```

Those constraints offer dedicated filtering algorithms which are able to make deductions where a decomposed model would not. For instance, constraint `alldifferent`(a, b, c, d) with $a, b \in [1, 4]$ and $c, d \in [3, 4]$ allows to deduce that a and b cannot be instantiated to 3 or 4; such rule cannot be inferred by simple binary constraints.

The up-to-date list of global constraints available in Choco can be found within the Javadoc API. Most of these global constraints are also described in Section Constraints.

2.2.9 Scheduling constraints

See also scheduling application.

```
cumulative, disjunctive, geost, pack, preceding, precedenceReified.
```

2.2.10 Sequencing constraints

```
multiCostRegular, regular, stretchCyclic, stretchPath,
```

Chapter 3

The solver

To create a solver, one just needs to create a new object as follow:

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

By this, a Solver for Constraint Programming is created.

The solver gives an API to read a model. The reading of a model is compulsory and must be done after the entire definition of the model.

```
solver.read(model);
```

The reading is divided in 2 parts: variables reading and constraints reading.

3.1 Variables reading

The solver iterates over the variables of the Model to create solver-specific variables and domains (as defined in the model). Thus, three types of variables can be created: integer variables, real variables and set variables. Depending on the constructor, the correct domain is created (like bounded domain or enumerated domain for integer variables).

> **Bound variables** are related to large domains which are only represented by their lower and upper bounds. The domain is encoded in a space efficient way and propagation events only concern bound updates. Value removals between the bounds are therefore ignored (*holes* are not considered). The level of consistency achieved by most constraints on these variables is called *bound-consistency*.
> On the contrary, the domain of an **enumerated variable** is explicitly represented and every value is considered while pruning. Basic constraints are therefore often able to achieve *arc-consistency* on enumerated variables (except for NP global constraint such as the cumulative constraint). Remember that switching from an enumerated variable to a bounded variables decrease the level of propagation achieved by the system.

Model variables and Solver variables are distinct. Solver variables are solver representation of the model variables. One can't access to variable value directly from the model variable. To access to a model variable thanks to the solver, use the following `Solver` API: `getVar(Variable v);`

3.1.1 Solver and IntegerVariables

A model integer variable can be accessed by the method `getVar(IntegerVariable v)` which returns a `IntDomainVar` object:

```
IntegerVariable x = makeEnumIntVar("x", 1, 100); // model variable
IntDomainVar xOnSolver = solver.getVar(x); // solver variable
```

The state of an `IntDomainVar` can be accessed through the main following public methods :

`IntDomainVar` API	description
`hasEnumeratedDomain()`	checks if the variable is an enumerated or a bound one
`getInf()`	returns the lower bound of the variable
`getSup()`	returns the upper bound of the variable
`getVal()`	returns the value if it is instantiated
`isInstantiated()`	checks if the domain is reduced to a singleton
`canBeInstantiatedTo(int v)`	checks if the value v is contained in the domain of the variable
`getDomainSize()`	returns the current size of the domain

For more informations on advanced uses of such `IntDomainVar`, see advanced uses.

3.1.2 Solver and SetVariables

A model set variable can be access by the method `getVar(SetVariable v)` which returns a `SetVar` object:

```
SetVariable x = makeBoundSetVar("x", 1, 40); // model variable
SetVar xOnSolver = solver.getVar(x); // solver variable
```

A set variable on integer values between $[1, n]$ has 2^n values (every possible subsets of $\{1..n\}$). This makes an exponential number of values and the domain is represented with two bounds corresponding to the intersection of all possible sets (called the kernel) and the union of all possible sets (called the envelope) which are the possible candidate values for the variable.

The state of a `SetVar` can be accessed through the main following public methods on the SetVar class:

`SetVar` API	description
`getCard()`	returns the `IntDomainVar` representing the cardinality of the set variable
`isInDomainKernel(int v)`	checks if value v is contained in the current kernel
`isInDomainEnveloppe(int v)`	checks if value v is contained in the current envelope
`getDomain()`	returns the domain of the variable as a `SetDomain`. Iterators on envelope or kernel can than be called
`getKernelDomainSize()`	returns the size of the kernel
`getEnveloppeDomainSize()`	returns the size of the envelope
`getEnveloppeInf()`	returns the first available value of the envelope
`getEnveloppeSup()`	returns the last available value of the envelope
`getKernelInf()`	returns the first available value of the kernel
`getKernelSup()`	returns the last available value of the kernel
`getValue()`	returns a table of integers `int[]` containing the current domain

For more informations on advanced uses of such `SetVar`, see advanced uses.

3.1.3 Solver and RealVariables

> *Real variables are still under development but can be used to solve toy problems such as small systems of equations.*

A model real variable can be access by the method `getVar(RealVariable v)` which returns a `RealVar` object:

```
RealVariable x = makeRealVar("x", 1.0, 3.0); // model variable
RealVar xOnSolver = s.getVar(x); // solver variable
```

Continuous variables are useful for non linear equation systems which are encountered in physics for example.

RealVar API	description
getInf()	returns the lower bound of the variable (double)
getSup()	returns the upper bound of the variable (double)
isInstantiated()	checks if the domain of a variable is reduced to a canonical interval. A canonical interval indicates that the domain has reached the precision given by the user or the solver

For more informations on advanced uses of such RealVar, see advanced uses.

3.2 Constraints reading

After variables, the Solver iterates over the constraints added to the Model. It creates Solver constraints that encapsulates a filtering algorithm which are called when a propagation step occur or when external events happen on the variables belonging to the constraint, such as value removals or bounds modifications. And it add it to the constraint network.

3.3 Search Strategy

A key ingredient of any constraint approach is a clever branching strategy. The construction of the search tree is done according to a series of Branching objects (that plays the role of achieving intermediate goals in logic programming). The user may specify the sequence of branching objects to be used to build the search tree. A common way to branch in CP is by assigning variables to values (such a branching is called AssignVar in choco). We will present in this section how to define your branching strategies with existing variables and values selectors.

3.3.1 Why is it important to define a search strategy ?

Once a fix point is reached, the Solver needs to select a variable and its value to continue the search. The way variables and values are chosen has a **real impact on the resolution step efficient**.

> *The search strategy should not be overlooked!!* An adapted search strategy can reduce: the execution time, the number of node expanded, the number of backtrack done.

Let see that small example:

```
Model m = new CPModel();
int n = 1000;
IntegerVariable var = makeBoundIntVar("var", 0, 2);
IntegerVariable[] bi = makeEnumIntVarArray("b", n, 0, 1);
m.addConstraint(eq(var, sum(bi)));

Solver badStrat = new CPSolver();
badStrat.read(m);
badStrat.setVarIntSelector(new MinDomain(badStrat));
badStrat.setValIntIterator(new IncreasingDomain());
badStrat.solve();
badStrat.printRuntimeSatistics();

Solver goodStrat = new CPSolver();
goodStrat.read(m);
goodStrat.setVarIntSelector(new MinDomain(goodStrat,
                                    goodStrat.getVar(new IntegerVariable[]{var})));
```

```
goodStrat.setValIntIterator(new DecreasingDomain());
goodStrat.solve();
goodStrat.printRuntimeSatistics();
```

This model ensures that $var = b_0 + b_1 + \ldots + b_{1000}$ where var has a small domain and b_i is a binary variable. The propagation has no effect on any domain and a fix point is reached at the beginning of the search. So, a decision has to be done choosing a variable and its value. As the default variable selector is `MinDomain` (see below), the solver will iterate over the variables, starting by the 1000 binary variables and ending with var, and 1001 nodes will be created.

3.3.2 Variable and value selection

A common way to explore the search tree in CP is by assigning values to variables. The branching class of Choco dedicated to this kind of search is **AssignVar**. More complex branching schemes can be performed in Choco but this section lists the default strategies available for exploring the search tree by assigning variables and that can be used within an `AssignVar` branching. These strategies are called *variable and value selection heuristics*.

The heuristics available in Choco and the API for selecting a given heuristic depend on the type of the considered variables. As instance, for integer variables, the default branching heuristic used by Choco is to choose the variable with current minimum domain size first (`MinDomain`) and to take its values in increasing order (`IncreasingDomain`). Customizing the value and variable heuristics on the integer variables of the solver can be done (before calling the `solve()` method) using the `Solver` API, as shown in the following example:

```
// select the next branching variable randomly
solver.setVarIntSelector(new RandomIntVarSelector(solver));
// select the values in increasing order
solver.setValIntIterator(new DecreasingDomain());
// *OR* select the next value randomly
solver.setValIntSelector(new RandomIntValSelector());
```

Variable selector.

It defines the way to choose the non instantiated variable on which the next decision will be made. A variable selector can be set using the following API:

`Solver` API	Variable	Default strategy
`setVarIntSelector(VarSelector)`	Integer	MinDomain
`setVarRealSelector(RealVarSelector)`	Real	CyclicRealVarSelector
`setVarSetSelector(SetVarSelector)`	Set	MinDomSet

The variable selectors currently available in Choco are the following: to complete

Integer Variable Selector	description
`StaticVarOrder(IntDomainVar[])`	A heuristic selecting the first non instantiated variable in the given static order
`MinDomain(Solver, IntDomainVar[])`	A heuristic selecting the variable with smallest domain
`DomOverDeg(Solver, IntDomainVar[])`	A heuristic selecting the variable with smallest ration (domainSize / degree), the *degree* of a variable is the number of constraints linked to it.
`DomOverDynDeg(Solver, IntDomainVar[])`	A heuristic selecting the variable with smallest degree, the *degree* of a variable is the number of constraints linked to it that is not completely instanciated.
`DomOverWDeg(Solver, IntDomainVar[])`	see example.
`MostConstrained(final Solver, final IntDomainVar[])`	A heuristic selecting the variable with the maximum degree
`RandomIntVarSelector(Solver, IntDomainVar[], long)`	A heuristic selecting randomly the non instantiated variable
`CompositeIntVarSelector(ConstraintSelector, HeuristicIntVarSelector)`	Composes two heuristics for selecting a variable: a first heuristic is applied for selecting a constraint. From that constraint a second heuristic is applied for selecting the variable
`LexIntVarSelector(HeuristicIntVarSelector, HeuristicIntVarSelector)`	applies two heuristics lexicographically for selecting a variable: a first heuristic is applied finding the best constraint, ties are broken with the second heuristic

Set Variable Selector	description
`MinDomSet(Solver)`	A heuristic selecting the variable with the smallest domain
`RandomSetVarSelector(Solver, SetVar[], long)`	A heuristic selecting randomly the variable
`StaticSetVarOrder(SetVar[])`	A heuristic selecting the first non instantiated variable in the given static order

Real Variable Selector	description
`CyclicRealVarSelector(Solver solver)`	Since a dichotomy algorithm is used, cyclic assiging is nedded for instantiate a real interval variable. A variable is selected several times to split its domain until it reaches the desired precision

> Solver variables have to be specified (not Model variables).

Value iterator

Once the variable has been choose, the Solver has to compute its value. The first way to do it is to schedule the value once and give an iterator to the solver. It can be done using the following API:

Solver API	Variable	Default strategy
`setValIntIterator(ValIterator)`	Integer	IncreasingDomain
`setValRealIterator(RealValIterator)`	Real	RealIncreasingDomain
`setValSetIterator(ValIterator)`	Set	MinEnv

The value iterators currently available in Choco are the following: to complete

Integer Value Iterator	description
`DecreasingDomain()`	A heuristic selecting value from the upper bound to the lower bound
`IncreasingDomain()`	A heuristic selecting value from the lower bound to the upper bound

Real Value Iterator	description
`RealIncreasingDomain()`	A heuristic selecting value from the lower bound to the upper bound

Value selector

The second way to do it is to compute the following value at each call. It can be done using the following API:

Solver API	Variable	Default strategy
`setValIntSelector(ValSelector)`	Integer	(none: see *value iterator*)
`setValRealSelector(ValSelector)`	Real	(none: see *value iterator*)
`setValSetSelector(SetValSelector)`	Set	(none: see *value iterator*)

The value selectors currently available in Choco are the following: to complete

Integer Value Selector	description
`MaxVal()`	Selecting the highest value in the domain
`MidVal()`	Selecting the middle value in the domain
`MinVal()`	Selecting the lowest value in the domain
`RandomIntValSelector()`	Selecting randomly the value in the domain

Set Value Selector	description
`MinEnv(Solver)`	Selecting the lowest value in the envelope and not in the kernel of the domain

3.3.3 Building a sequence of branching object

You might want to apply different heuristics to different set of variables of the problem. In that case, the search is viewed as a sequence of branching objects (or goals). Up to now, we only had one branching or one goal including all the variables of the problem but several goals can be used.

Adding a new goal is made through the solver with the `addGoal(AbstractBranching b)` method. As for the addition of your own limit, dont call the `solve()` method, but instead: build the solver by yourself, add your sequence of branching, and call the `launch()` method of the solver.

The following example add three branching objects on integer variables *vars1*, *vars2* and set variables *svars* to solver *s*. The first two branchings are both `AssignVar` but use two different variable/values selection strategies:

```
s.attachGoal(new AssignVar(new MinDomain(s,s.getVar(vars1)), new IncreasingDomain()));
s.addGoal(new AssignVar(new DomOverDeg(s,s.getVar(vars2)),new DecreasingDomain());
s.addGoal(new AssignSetVar(new MinDomSet(s,s.getVar(svars)), new MinEnv(s)));
s.generateSearchStrategy();
s.launch();
```

An example of how to set the search solver in case of optimization in given in the tutorial on cumulative.

3.3.4 Dom/WDeg

to introduce

```
Solver s = new CPSolver();
s.read(model);

s.attachGoal(new DomOverWDegBranching(s, new IncreasingDomain()));
```

```
s.setFirstSolution(true);
s.generateSearchStrategy();
```

The decision variables can be set using :

```
DomOverWDegBranching dwdeg = new DomOverWDegBranching(s, new IncreasingDomain());
dwdeg.setBranchingVars(vars);
```

3.3.5 Impacts

to introduce

```
CPSolver s = new CPSolver();
s.read(model);

//create the branching on the decision variables vars.
ImpactBasedBranching ibb = new ImpactBasedBranching(s, vars);

//initialize the impacts with a time limit of 10s
ibb.getImpactStrategy().initImpacts(10000);

s.generateSearchStrategy();
s.attachGoal(ibb);
s.setFirstSolution(true);
s.launch();
```

3.3.6 Restarts

You can set geometric restarts by using the following API available on the solver:

```
setGeometricRestart(int base, double grow);
setGeometricRestart(int base, double grow, int restartLimit);
```

It performs a search with restarts regarding the number of backtrack. An initial allowed number of backtrack is given (parameter base) and once this limit is reached a restart is performed and the new limit imposed to the search is increased by multiplying the previous limit with the parameter grow. restartLimit parameter states the maximum number of restarts. Restart strategies makes really sense with strategies that make choices based on the past experience of the search : `DomOverWdeg` or Impact based search. It could also be used with a random heuristic

```
CPSolver s = new CPSolver();
s.read(model);

s.setGeometricRestart(14, 1.5d);
s.setFirstSolution(true);
s.generateSearchStrategy();
s.attachGoal(new DomOverWDegBranching(s, new IncreasingDomain()));
s.launch();
```

You can also set Luby restarts by using the following API available on the solver:

```
setLubyRestart(int base);
setLubyRestart(int base, int grow);
setLubyRestart(int base, int grow, int restartLimit);
```

it performs a search with restarts regarding the number of backtracks. One way to describe this strategy is to say that all run lengths are power of two, and that each time a pair of runs of a given length has been completed, a run of twice that length is immediatly executed. The limit is equals to *length*base*.

- **example with growing factor of 2 : [1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1,...]**
- **example with growing factor of 3 : [1, 1, 1, 3, 1, 1, 1, 3, 9,...]**

```
CPSolver s = new CPSolver();
s.read(model);

s.setLubyRestart(50, 2, 100);
s.setFirstSolution(true);
s.generateSearchStrategy();
s.attachGoal(new DomOverWDegBranching(s, new IncreasingDomain()));
s.launch();
```

3.4 Limiting Search Space

The Solver class provides some limits on the search strategy that you can fix or just monitor. Limits may be imposed on the search algorithm to avoid spending too much time in the exploration. The limits are updated and checked each time a new node is created. It has to be specified before the resolution. After having created the solver, you can specify whether or not you want to fix a limit:

time limit State a time limit on tree search. When the execution time is equal to the time limit, the search stops whatever a solution is found or not. You can define a time limit with the following API : `setTimeLimit(int timeLimit)` where unit is millisecond. Or just monitor (or not) the search time with the API : `monitorTimeLimit(boolean b)`. The default value is set to `true`. Finally, you can get the time limit, once the solve method has been called, with the API: `getTimeCount()`

node limit State a node limit on tree search. When the number of nodes explored is equal to the node limit, the search stops whatever a solution is found or not. You can define a node limit with the following API: `setNodeLimit(int nodeLimit)` where unit is the number of nodes. Or just monitor (or not) the number of nodes explored with the API: `monitorNodeLimit(boolean b)`. The default value is set to `true`. Finally, you can get the node limit, once the solve method has been called, with the API: `getNodeCount()`

backtrack limit State a backtrack limit on tree search. When the number of backtracks done is equal to the backtrack limit, the search stops whatever a solution is found or not. You can define a backtrack limit with the following API: `setBackTrackLimit(int backtrackLimit)` where unit is the number of backtracks. Or just monitor (or not) the number of backtrack done with the API: `monitorBackTrackLimit(boolean b)`. The default value is set to `false`. Finally, you can get the backtrack limit, once the solve method has been called, with the API: `getBackTrackCount()`

fail limit State a fail limit on tree search. When the number of failure is equal to the fail limit, the search stops whatever a solution is found or not. You can define a fail limit with the following API : `setFailLimit(int failLimit)` where unit is the number of failure. Or just monitor (or not) the number of failure encountered with the API : `monitorFailLimit(boolean b)`. The default value is set to `false`. Finally, you can get the fail limit, once the solve method has been called, with the API : `getFailCount()`

CPU time limit State a CPU limit on tree search. When the CPU time (user + system) is equal to the CPU time limit, the search stops whatever a solution is found or not. You can define a CPU time limit with the following API: `setCpuTimeLimit(int timeLimit)` where unit is millisecond. Or just monitor (or not) the search time with the API: `monitorCpuTimeLimit(boolean b)`. The default value is set to `false`. Finally, you can get the CPU time limit, once the solve method has been called, with the API: `getCpuTimeCount()`

add example

3.5 Solve a problem

As Solver is the second element of a Choco program, the control of the search process without using predefined tools is made on the Solver.

Solver API	description
`solve()`	Compute the first solution of the Model, if the Model is feasible.
`solve(boolean all)`	If *all* is set to true, computes all solutions of the Model, if the Model is feasible.
`solveAll()`	Computes all the solution of the Model, if the Model is feasible.
`propagate()`	Computes initial propagation of the Model, and reachs the first Fix Point. It reduces variables Domain through constraints linked and other variables domain. Can throw a `ContradictionException` if the Solver detects a contradiction in the Model.
`maximize(Var obj, boolean restart)`	Allows user to find a solution that maximizing the objective varible *obj*. The optimization finds a first solution then finds a new solution that improves *obj* and so on till no other solution can be found that improves *obj*. Parameter *restart* is a boolean indicating whether the Solver will restart the search after each solution found (if set to `true`) or if it will keep backtracking from the leaf of the last solution found. See example. **Beware**: the variable *obj* expected must be a Solver variable and not a Model variable.
`minimize(Var obj, boolean restart)`	Allows user to find a solution that minimizing the objective varible *obj*. The optimization finds a first solution then finds a new solution that improves *obj* and so on till no other solution can be found that improves *obj*. Parameter *restart* is a boolean indicating whether the Solver will restart the search after each solution found (if set to `true`) or if it will keep backtracking from the leaf of the last solution found. See example. **Beware**: the variable *obj* expected must be a Solver variable and not a Model variable.
`nextSolution()`	Allows the Solver to find the next solution, if one or more solution have already been find with `solve()` or `nextSolution()`.
`isFeasible()`	Indicates whether or not the Model has at least one solution.

3.5.1 Solver settings

Logs

The solver class is instrumented in order to produce trace statements throughout search. The verbosity level of the solver can be set, by the following static method

```
CPSolver.setVerbosity(CPSolver.SEARCH);
// And after solver.solve()
CPSolver.flushLogs();
```

The code above ensure that messages are printed in order to describe the construction of the search tree.

Five verbosity levels are available:

Level	prints...
`CPSolver.SILENT`	nothing
`CPSolver.SOLUTION`	messages whenever a solution is reached
`CPSolver.SEARCH`	a message at each choice point
`CPSolver.PROPAGATION`	messages to trace propagation
`CPSolver.FINEST`	high level messages to trace propagation

Note, that in the case of a verbosity set to `CPSolver.SEARCH`, trace statements are printed up to a maximal depth in the search tree. By default, only the 5 first levels are traced, but you can change the value of this threshold, say to 10, with the following setter method:

```
solver.setLoggingMaxDepth(10);
```

3.5.2 Optimization

to introduce

```
Model m = new CPModel();
IntegerVariable obj1 = makeEnumIntVar("obj1", 0, 7);
IntegerVariable obj2 = makeEnumIntVar("obj1", 0, 5);
IntegerVariable obj3 = makeEnumIntVar("obj1", 0, 3);
IntegerVariable cost = makeBoundIntVar("cout", 0, 1000000);
int capacity = 34;
int[] volumes = new int[]{7, 5, 3};
int[] energy = new int[]{6, 4, 2};
// capacity constraint
m.addConstraint(leq(scalar(volumes, new IntegerVariable[]{obj1, obj2, obj3}), capacity));

// objective function
m.addConstraint(eq(scalar(energy, new IntegerVariable[]{obj1, obj2, obj3}), cost));

Solver s = new CPSolver();
s.read(m);

s.maximize(s.getVar(cost), false);
```

Chapter 4

Advanced uses of Choco

4.1 Define your own limit search space

To define your own limits/statistics (notice that a limit object can be used only to get statistics about the search), you can create a limit object by extending the `AbstractGlobalSearchLimit` class or implementing directly the interface `IGlobalSearchLimit`. Limits are managed at each node of the tree search and are updated each time a node is open or closed. Notice that limits are therefore time consuming. Implementing its own limit need only to specify to the following interface :

```
/**
 * The interface of objects limiting the global search exploration
 */
public interface GlobalSearchLimit {

  /**
   * resets the limit (the counter run from now on)
   * @param first true for the very first initialization, false for subsequent ones
   */
  public void reset(boolean first);

  /**
   * notify the limit object whenever a new node is created in the search tree
   * @param solver the controller of the search exploration, managing the limit
   * @return true if the limit accepts the creation of the new node, false otherwise
   */
  public boolean newNode(AbstractGlobalSearchSolver solver);

  /**
   * notify the limit object whenever the search closes a node in the search tree
   * @param solver the controller of the search exploration, managing the limit
   * @return true if the limit accepts the death of the new node, false otherwise
   */
  public boolean endNode(AbstractGlobalSearchSolver solver);
}
```

Look at the following example to see a concrete implementation of the previous interface. We define here a limit on the depth of the search (which is not found by default in choco). The `getWorldIndex()` is used to get the current world, i.e the current depth of the search or the number of choices which have been done from baseWorld.

```
public class DepthLimit extends AbstractGlobalSearchLimit {

  public DepthLimit(AbstractGlobalSearchSolver theSolver, int theLimit) {
    super(theSolver,theLimit);
    unit = "deep";
```

```
    }

    public boolean newNode(AbstractGlobalSearchSolver solver) {
      nb = Math.max(nb, this.getProblem().getWorldIndex()
      this.getProblem().getSolver().getSearchSolver().baseWorld);
      return (nb < nbMax);
    }

    public boolean endNode(AbstractGlobalSearchSolver solver) {
      return true;
    }

    public void reset(boolean first) {
     if (first) {
      nbTot = 0;
     } else {
      nbTot = Math.max(nbTot, nb);
     }
     nb = 0;
    }
```

Once you have implemented your own limit, you need to tell the search solver to take it into account. Instead of using a call to the `solve()` method, you have to create the search solver by yourself and add the limit to its limits list such as in the following code :

```
Solver s = new CPSolver();
s.read(model);
s.setFirstSolution(true);
s.generateSearchStrategy();
s.getSearchStrategy().limits.add(new DepthLimit(s.getSearchStrategy(),10));
s.launch();
```

4.2 Define your own search strategy

Section Search strategy presented the default branching strategies available in Choco and showed how to post them or to compose them as goals. In this section, we will start with a very simple and common way to branch by choosing values for variables and specially how to define its own variable/value selection strategy. We will then focus on more complex branching such as dichotomic or n-ary choices. Finally we will show how to control the search space in more details with well known strategy such as LDS (Limited discrepancy search).

For integer variables, the variable and value selection strategy objects are based on the following interfaces:

- `IntVarSelector` : Interface for the (integer) variable selection
- `ValIterator` : Interface to describes an iteration scheme on the domain of a variable
- `ValSelector` : Interface for a value selection

Concrete examples of these interfaces are respectively `DomOverDeg`, `IncreasingDomain`, `MinVal`. The default branchings currently supported by Choco are available in packages `src.choco.cpsolver.search.integer` for integer variables, `src.choco.cpsolver.search.set` for set variables, `src.choco.cpsolver.search.real` for real variables.

4.2.1 Define your own variable selection

You may extend this small library of branching schemes and heuristics by defining your own concrete classes of `AbstractIntVarSelector`. We give here an example of an `IntVarSelector` with the implementation of a static variable ordering :

```
public class StaticVarOrder extends AbstractIntVarSelector {

    // the sequence of variables that need be instantiated
   protected IntDomainVar[] vars;

   public StaticVarOrder(IntDomainVar[] vars) {
        this.vars = vars;
   }

   public IntDomainVar selectIntVar() {
        for (int i = 0; i < vars.length; i++)
            if (!vars[i].isInstantiated())
                return vars[i];
        return null;
   }
}
```

Notice on this example that you only need to implement method `selectIntVar()` which belongs to the contract of `IntVarSelector`. This method should return a non instantiated variable or `null`. Once the branching is finished, the next branching (if one exists) is taken by the search algorithm to continue the search, otherwise, the search stops as all variable are instantiated. To avoid the loop over the variables of the branching, a backtrackable integer (`StoredInt`) could be used to remember the last instantiated variable and to directly select the next one in the table. Notice that backtrackable structures could be used in any of the code presented in this chapter to speedup the computation of dynamic choices.

You can add your variable selector to the solver as common variable selector, using the `Solver` API:

```
solver.setVarIntSelector(new MyVarSelector(...));
```

4.2.2 Define your own value selection

You may also define your own concrete classes of `ValIterator` or `ValSelector`.

Value selector

We give here an example of an `IntValSelector` with the implementation of a minimum value selecting:

```
public class MinVal extends AbstractSearchHeuristic implements ValSelector {
  /**
    * selecting the lowest value in the domain
    * @param x the variable under consideration
    * @return what seems the most interesting value for branching
    */
   public int getBestVal(IntDomainVar x) {
        return x.getInf();
   }
}
```

Only `getBestVal()` method must be implemented, returning the best value *in the domain* according to the heuristic.

You can add your value selector to the solver as common variable selector, using the `Solver` API:

```
solver.setValIntSelector(new MyValSelector(...));
```

Values iterator

We give here an example of an `ValIterator` with the implementation of an increasing domain iterator:

```
public final class IncreasingDomain implements ValIterator {
   /**
```

```
     * testing whether more branches can be considered after branch i,
      * on the alternative associated to variable x
     * @param x the variable under scrutiny
     * @param i the index of the last branch explored
     * @return true if more branches can be expanded after branch i
     */
      public boolean hasNextVal(Var x, int i) {
          return (i < ((IntDomainVar) x).getSup());
      }

    /**
     * accessing the index of the first branch for variable x
     * @param x the variable under scrutiny
     * @return the index of the first branch: first value to be assigned to x
     */
      public int getFirstVal(Var x) {
          return ((IntDomainVar) x).getInf();
      }

    /**
     * generates the index of the next branch after branch i,
      * on the alternative associated to variable x
     * @param x the variable under scrutiny
     * @param i the index of the last branch explored
     * @return the index of the next branch to be expanded after branch i
     */
      public int getNextVal(Var x, int i) {
          return ((IntDomainVar) x).getNextDomainValue(i);
      }
  }
```

You can add your value iterator to the solver as common variable selector, using the `Solver` API:

```
s.setValIntIterator(new MyValIterator(..));
```

4.2.3 How does a search loop work ?

The seach loop is created when a `solve()` method is called. It goes down and up in the branches in order to cover the tree search.

Algorithm of the search loop in Choco

```
next_move = new node
WHILE no solution AND in search limit
      IF next_move is new node
     THEN
         create a new node : variable/value selection ;
         IF node exists
         THEN
             next_move <-- go down branch ;
         ELSE
             next_move <-- go up branch ;
             solution is found ;

     ELSE IF next_move is go down branch
         propagate ;
         IF no contradiction
         THEN
             next_move <-- new node ;
         ELSE
             next_move <-- go up branch ;
```

```
    ELSE IF next_move is go up branch
        find next branch ;
        propagate ;
        IF has next branch AND no contradiction
        THEN
            next_move <-- go down branch ;
        ELSE
            next_move <-- go up branch ;

    END IF

END WHILE
```

4.2.4 How to define your own Branching object

Beyond Variable/value selection...

under development See old version

4.3 Define your own constraint

This section describes how to add you own constraint, with specific propagation algorithms. Note that this section is only useful in case you want to express a constraint for which the basic propagation algorithms (using tables of tuples, or boolean predicates) are not efficient enough to propagate the constraint.

The general process consists in defining a new constraint class and implementing the various propagation methods. We recommend the user to follow the examples of existing constraint classes (for instance, such as `GreaterOrEqualXYC` for a binary inequality)

4.3.1 The constraint hierarchy

Each new constraint must be represented by an object implementing the **SConstraint** interface (`S` for solver constraint). To help the user defining new constraint classes, several abstract classes defining `SConstraint` have been implemented. These abstract classes provide the user with a management of the constraint network and the propagation engineering. They should be used as much as possible.

For constraints on integer variables, the easiest way to implement your own constraint is to inherit from one of the following classes, depending of the number of solver integer variables (`IntDomainVar`) involved:

Default class to implement	number of solver integer variables
`AbstractUnIntSConstraint`	**one** variable
`AbstractBinIntSConstraint`	**two** variables
`AbstractTernIntSConstraint`	**three** variables
`AbstractLargeIntSConstraint`	any number of variables.

Constraints over integers must implement the following methods (grouped in the `IntSConstraint` interface):

Method to implement	description
`pretty()`	Returns a pretty print of the constraint
`propagate()`	The main propagation method (propagation from scratch). Propagating the constraint until local consistency is reached.
`awake()`	Propagating the constraint for the very first time until local consistency is reached. The awake is meant to initialize the data structures contrary to the propagate. Specially, it is important to avoid initializing the data structures in the constructor.
`awakeOnInst(int x)`	Default propagation on instantiation: full constraint repropagation.
`awakeOnBounds(int x)`	Default propagation on improved bounds: propagation on domain revision.
`awakeOnRemovals(int x, IntIterator v)`	Default propagation on mutliple values removal: propagation on domain revision. The iterator allow to iterate over the values that have been removed.
Methods `awakeOnBounds` and `awakeOnRemovals` can be replaced by more fine grained methods:	
`awakeOnInf(int x)`	Default propagation on improved lower bound: propagation on domain revision.
`awakeOnSup(int x)`	Default propagation on improved upper bound: propagation on domain revision.
`awakeOnRem(int x, int v)`	Default propagation on one value removal: propagation on domain revision.
To use the constraint in expressions or reification, the following minimum API is mandatory:	
`isSatisfied(int[] x)`	Tests if the constraint is satisfied when the variables are instantiated.
`isEntailed()`	Checks if the constraint must be checked or must fail. It returns true if the constraint is known to be satisfied whatever happend on the variable from now on, false if it is violated.
`opposite()`	It returns an AbstractSConstraint that is the opposite of the current constraint.

In the same way, a **set constraint** can inherit from `AbstractUnSetSConstraint`, `AbstractBinSetSConstraint`, `AbstractTernSetSConstraint` or `AbstractLargeSetSConstraint`.

A **real constraint** can inherit from `AbstractUnRealSConstraint`, `AbstractBinRealSConstraint` or `AbstractLargeRealSConstraint`.

A mixed constraint between **set and integer variables** can inherit from `AbstractBinSetIntSConstraint` or `AbstractLargeSetIntSConstraint`.

A simple way to implement its own constraint is to:

- create an empty constraint with only `propagate()` method implemented and every `awakeOnXxx()` ones set to `this.constAwake(false);`
- when the propagation filter is sure, separate it into the `awakeOnXxx()` methods in order to have finer granularity
- finally, if necessary, use backtrackables objects to improve the efficient of your constraint

How do I add my constraint to the Model ?

Adding your constraint to the model requires you to definite a specific constraint manager (that can be a inner class of your Constraint). This manager need to implement:

```
makeConstraint(Solver s, Variable[] vars, Object params, HashSet<String> options)
```

This method allows the Solver to create an instance of your constraint, with your parameters and Solver objects.

> If you create your constraint manager as an inner class, you must declare this class as **public and static**. If you don't, the solver can't instantiate your manager.

Once this manager has been implemented, you simply add your constraint to the model using the `addConstraint()` API with a `ComponentConstraint` object:

```
model.addConstraint( new ComponentConstraint(MyConstraintManager.class, params, vars) );
// OR
model.addConstraint( new ComponentConstraint("package.of.MyConstraint", params, vars) );
```

Where *params* is whatever you want (`Object[], int, String`,...) and *vars* is an array of Model Variables (or more specific) objects.

4.3.2 Example: implement and add the `IsOdd` constraint

One creates the constraint by implementing the `AbstractUnIntSConstraint` (one integer variable) class:

```
public class IsOdd extends AbstractUnIntSConstraint {

        public IsOdd(IntDomainVar v0) {
            super(v0);
        }

         /** Default initial propagation: full constraint re-propagation */
         public void awake() throws ContradictionException {
            DisposableIntIterator it = v0.getDomain().getIterator();
             while(it.hasNext()){
                 int val = it.next();
                 if(val%2==0)
                     v0.removeVal(val, 0);
             }
        }

        /** Propagation until local consistency is reached */
        public void propagate() throws ContradictionException {
            if(v0.isInstantiated() && v0.getVal()%2==0)
                fail();
        }
    }
```

To add the constraint to the model, one creates the following class (or inner class):

```
public static class IsOddManager implements IntConstraintManager {
    public SConstraint makeConstraint(Solver s, Variable[] vars, Object params, HashSet<String
        > options) {
        return new IsOdd(s.getVar((IntegerVariable)vars[0]));
    }
}
```

It calls the constructor of the constraint, with every *vars*, *params* and *options* needed.

Then, the constraint can be added to a model as follows:

```
// Creation of the model
Model m = new CPModel();

// Declaration of the variable
IntegerVariable aVar = Choco.makeIntVar("a_variable", 0, 10);

// Adding the constraint to the model, 1st solution:
m.addConstraint(new ComponentConstraint(IsOddManager.class, null, new IntegerVariable[]{aVar
    }));
// OR 2nd solution:
m.addConstraint(new ComponentConstraint("myPackage.Constraint.IsOddManager", null, new
    IntegerVariable[]{aVar}));

Solver s = new CPSolver();
s.read(m);
s.solve();
```

And that's it!!

4.3.3 Example of an empty constraint

See the complete code: ConstraintPattern.zip

```
public class ConstraintPattern extends AbstractLargeIntSConstraint {

    public ConstraintPattern(IntDomainVar[] vars) {
        super(vars);
    }

    /**
    * pretty print. The String is not constant and may depend on the context.
    * @return a readable string representation of the object
    */
    public String pretty() {
        return null;
    }

    /**
    * check whether the tuple satisfies the constraint
    * @param tuple values
    * @return true if satisfied
    */
    public boolean isSatisfied(int[] tuple) {
        return false;
    }

    /**
    * propagate until local consistency is reached
    */
    public void propagate() throws ContradictionException {
        // elementary method to implement
    }

    /**
    * propagate for the very first time until local consistency is reached.
    */
    public void awake() throws ContradictionException {
        constAwake(false); // change if necessary
    }
```

```
    /**
    * default propagation on instantiation: full constraint re-propagation
    * @param var index of the variable to reduce
    */
    public void awakeOnInst(int var) throws ContradictionException {
        constAwake(false); // change if necessary
    }

    /**
    * default propagation on improved lower bound: propagation on domain revision
    * @param var index of the variable to reduce
    */
    public void awakeOnInf(int var) throws ContradictionException {
        constAwake(false); // change if necessary
    }


    /**
    * default propagation on improved upper bound: propagation on domain revision
    * @param var index of the variable to reduce
    */
    public void awakeOnSup(int var) throws ContradictionException {
        constAwake(false); // change if necessary
    }

    /**
    * default propagation on improve bounds: propagation on domain revision
    * @param var index of the variable to reduce
    */
    public void awakeOnBounds(int var) throws ContradictionException {
        constAwake(false); // change if necessary
    }

    /**
    * default propagation on one value removal: propagation on domain revision
    * @param var index of the variable to reduce
    * @param val the removed value
    */
    public void awakeOnRem(int var, int val) throws ContradictionException {
        constAwake(false); // change if necessary
    }

    /**
    * default propagation on one value removal: propagation on domain revision
    * @param var index of the variable to reduce
    * @param delta iterator over remove values
    */
    public void awakeOnRemovals(int var, IntIterator delta) throws ContradictionException {
        constAwake(false); // change if necessary
    }
}
```

The first step to create a constraint in Choco is to implement all `awakeOn...` methods with `constAwake(false)` and to put your propagation algorithm in the `propagate()` method.

A constraint can choose not to react to fine grained events such as the removal of a value of a given variable but instead delay its propagation at the end of the fix point reached by "fine grained events" and fast constraints that deal with them incrementally (that's the purpose of the constraints events queue).

To do that, you can use `constAwake(false)` that tells the solver that you want this constraint to be called only once the variables events queue is empty. This is done so that heavy propagators can

delay their action after the fast one to avoid doing a heavy processing at each single little modification of domains.

4.4 Define your own operator

to complete

4.5 Define your own variable

to complete

4.6 Backtrackable structures

to complete

4.7 Logging System

Choco logging system is based on the `java.util.logging` package and located in the package `common.logging`. Most Choco abstract classes or interfaces propose a static field `LOGGER`. The following figures present the architecture of the logging system with the default verbosity.

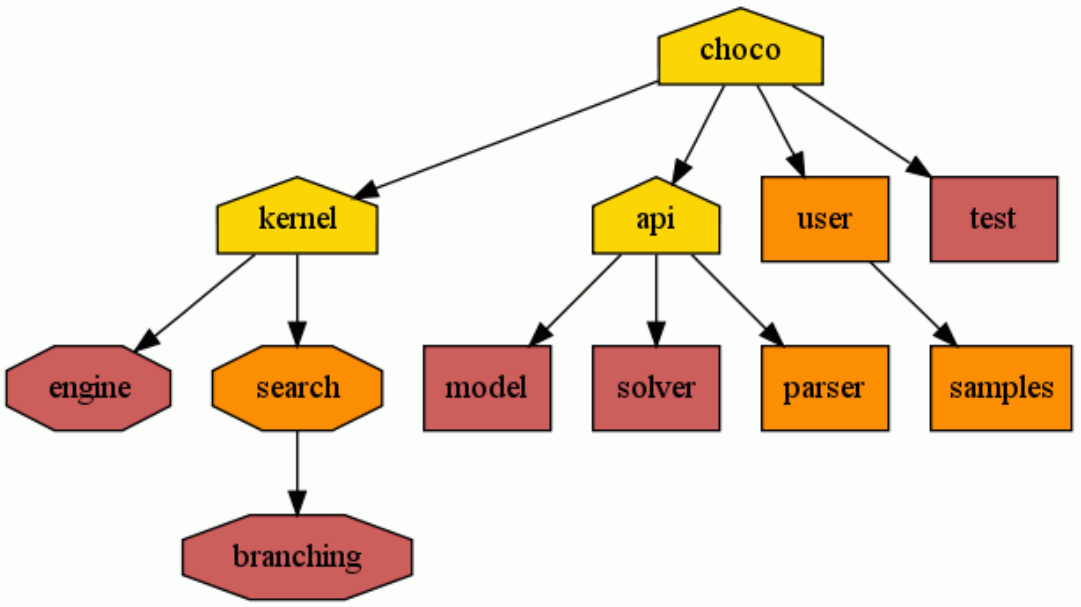

Figure 4.1: Logger Tree with the default verbosity

The shape of the node depicts the type of logger:

- The *house* loggers represent private loggers. Do not use directly these loggers because their level are low and all messages would always be displayed.
- The *octagon* loggers represent critical loggers. These loggers are provided in the variables, constraints and search classes and could have a huge impact on the global performances.
- The *box* loggers are provided for dev and users.

The color of the node gives its logging level with DEFAULT verbosity: `Level.FINEST` (gold), `Level.INFO` (orange), `Level.WARNING` (red).

Verbosity.

The following table summarizes the verbosity available in choco:

Verbosity	level	description
OFF	0	disable logging
SILENT	1	display only severe messages
DEFAULT	2	display warning and some API messages
VERBOSE	3	display more API messages and a solving message
SOLUTION	4	display all solutions
SEARCH	5	display the search tree
FINEST	6	display all logs

More precisely, if the verbosity level is greater than DEFAULT, then the verbosity levels of the model and of the solver are increased to INFO, and the verbosity levels of the search and of the branching are slightly modified to display the solution(s) and search messages.

The verbosity level can be changed as follows:

```
ChocoLogging.setVerbosity(Verbosity.VERBOSE);
```

How to write logging statements ?

- Critical Loggers are provided to display error or warning. Displaying too much message really **impacts the performances**.
- Check the logging level before creating arrays or strings.
- Avoid multiple calls to `Logger` functions. Prefer to build a `StringBuilder` then call the `Logger` function.
- Use the `Logger.log` function instead of building string in `Logger.info()`.

Handlers.

Logs are displayed on `System.out` but warnings and severe messages are also displayed on `System.err`. `ChocoLogging.java` also provides utility functions to easily change handlers:

- Functions `set...Handler` remove current handlers and replace them by a new handler.
- Functions `add...Handler` add new handlers but do not touch existing handlers.

Define your own logger.

```
ChocoLogging.makeUserLogger(String suffix);
```

Chapter 5

Applications with global constraints

5.1 Scheduling and use of the cumulative constraint

This tutorial is the Choco2 version of this one

We present a simple example of a scheduling problem solved using the cumulative global constraint.

The problem is to maximize the number of tasks that can be scheduled on a single resource within a given time horizon.

The following picture summarizes the instance that we will use as example. It shows the resource profile on the left and on the right, the set of tasks to be scheduled. Each task is represented here as a rectangle whose height is the resource consumption of the task and whose length is the duration of the task. Notice that the profile is not a straight line but varies in time.

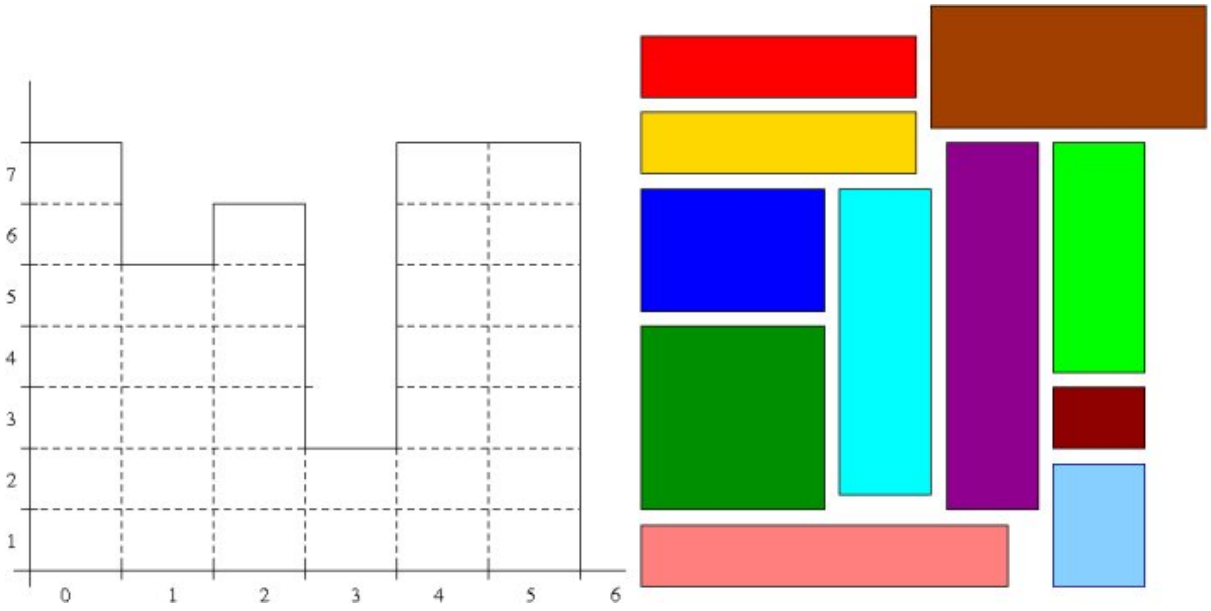

Figure 5.1: A cumulative scheduling problem instance

This tutorial might help you to deal with :

- The profile of the resource that varies in time whereas the API of the cumulative only accepts a constant capacity
- The objective function that implies optional tasks which is a priori not allowed by cumulative
- A search heuristic that will first assign the tasks to the resource and then try to schedule them while maximizing the number of tasks

The first point is easy to solve by adding fake tasks at the right position to simulate the consumption of the resource. The second point is possible thanks to the ability of the cumulative to handle variable heights. We shall explain it in more details soon.

Let's have a look at the source code and first start with the representation of the instance. We need three fake tasks to decrease the profile accordindly to the instance capacity. There is otherwise 11 tasks. Their heights and duration are fixed and given in the two following int[] tables. The three first tasks correspond to the fake one are all of duration 1 and heights 2, 1, 4.

```
CPModel m = new CPModel();
// data
int n = 11 + 3; //number of tasks (include the three fake tasks)
int[] heights_data = new int[]{2, 1, 4, 2, 3, 1, 5, 6, 2, 1,3, 1, 1, 2};
int[] durations_data = new int[]{1, 1, 1, 2, 1, 3, 1, 1, 3, 4,2, 3, 1, 1};
```

The variables of the problem consist of four variables for each task (start, end, duration, height). We recall here that the scheduling convention is that a task is active on the interval [start, end-1] so that the upper bound of the start and end variables need to be 5 and 6 respectively. Notice that start and end variables are BoudIntVar variables. Indeed, the cumulative is only performing bound reasonning so it would be a waste of efficiency to declare here EnumVariables. Duration and heights are constant in this problem. However, our plan is to simulate the allocation of a task to the resource by using variable height. In other word, we will define the height of the task i as a variable of domain $\{0, \texttt{height[i]}\}$. The height of the task takes its normal value if the task is assigned to the resource and 0 otherwise. The duration is really constant and is therefore created as a ConstantIntVar.

Moreover, we add a boolean variable per task to specify if the task is assigned to the resource or not. The objective variable is created as a BoundIntVar.

```
IntegerVariable capa = constant(7);
IntegerVariable[] starts = makeIntVarArray("start", n, 0, 5, "cp:bound");
IntegerVariable[] ends = makeIntVarArray("end", n, 0, 6, "cp:bound");

IntegerVariable[] duration = new IntegerVariable[n];
IntegerVariable[] height = new IntegerVariable[n];
for (int i = 0; i < height.length; i++) {
    duration[i] = constant(durations_data[i]);
    height[i] = makeIntVar("height " + i, new int[]{0, heights_data[i]});
}

IntegerVariable[] bool = makeIntVarArray("taskIn?", n, 0, 1);
IntegerVariable obj = makeIntVar("obj", 0, n, "cp:bound", "cp:objective");
```

We then add the constraints to the model. Three constraints are needed. First, the cumulative ensures that the resource consumption is respected at any time. Then we need to make sure that if a task is assigned to the cumulative, its height can not be null which is done by the use of boolean channeling constraints. Those constraints ensure that :

$$\texttt{bool[i]} = 1 \iff \texttt{height[i]} = \texttt{heights\_data[i]}$$

We state the objective function to be the sum of all boolean variables.

```
//post the cumulative
m.addConstraint(cumulative(starts, ends, duration, height, capa, ""));
//post the channeling to know if the task is scheduled or not
for (int i = 0; i < n; i++) {
    m.addConstraint(boolChanneling(bool[i], height[i], heights_data[i]));
}

//state the objective function
m.addConstraint(eq(sum(bool), obj));
```

Finally we fix the fake task at their position to simulate the profil:

```
CPSolver s = new CPSolver();
s.read(m);

//set the fake tasks to establish the profile capacity of the resource
try {
    s.getVar(starts[0]).setVal(1); s.getVar(ends[0]).setVal(2); s.getVar(height[0]).setVal(2);
    s.getVar(starts[1]).setVal(2); s.getVar(ends[1]).setVal(3); s.getVar(height[1]).setVal(1);
    s.getVar(starts[2]).setVal(3); s.getVar(ends[2]).setVal(4); s.getVar(height[2]).setVal(4);
} catch (ContradictionException e) {
    System.out.println("error, no contradiction expected at this stage");
}
```

We are now ready to solve the problem. We could call a maximize(false) but we want to add a specific heuristic that first assigned the tasks to the cumulative and then tries to schedule them.

```
s.maximize(s.getVar(obj),false);
```

We now want to print the solution and will use the following code :

```
System.out.println("Objective : " + (s.getVar(obj).getVal() - 3));
for (int i = 3; i < starts.length; i++) {
   if (s.getVar(height[i]).getVal() != 0)
      System.out.println("[" + s.getVar(starts[i]).getVal() + " - "
                         + (s.getVar(ends[i]).getVal() - 1) + "]:"
                         + s.getVar(height[i]).getVal());
}
```

Choco gives the following solution :

```
Objective : 9
[1 - 2]:2
[0 - 0]:3
[2 - 4]:1
[4 - 4]:5
[5 - 5]:6
[0 - 2]:2
[0 - 3]:1
[3 - 5]:1
[0 - 0]:1
```

This solution could be represented by the following picture :

Notice that the cumulative gives a necesserary condition for packing (if no schedule exists then no packing exists) but this condition is not sufficient as shown on the picture because it only ensures that the capacity is respected at each time point. Specially, the tasks might be splitted to fit in the profile as in the previous solution. The complete code can be found here.

5.2 Placement and use of the Geost constraint

The global constraint **geost**(k, O, S, C) handles in a generic way a variety of geometrical constraints C in space and time between polymorphic $k \in \mathbb{N}$ dimensional objects O, each of which taking a shape among a set of shapes S during a given time interval and at a given position in space. Each shape from S is defined as a finite set of shifted boxes, where each shifted box is described by a box in a k-dimensional space at the given offset with the given sizes.

More precisely a *shifted box* s= *shape(sid,t[],l[])* is an entity defined by a shape id *sid*, an shift offset $s.t[d]$, $0 \leq d < k$, and a size $s.l[d] > 0$, $0 \leq d < k$. All attributes of a shifted box are integer values. Then, a *shape* from S is a collection of shifted boxes sharing all the same shape id. Note that the shifted boxes associated with a given shape may or may not overlap. This sometimes allows a drastic reduction in the number of shifted boxes needed to describe a shape. Each *object* o= *object(id, sid,x[], start, duration,end)* from O is an entity defined by a unique object id $o.id$ (an integer), a shape id $o.sid$, an origin $o.x[d]$, $0 \leq d < k$, a starting time $o.start$, a duration $o.duration > 0$, and a finishingxs time $o.end$.

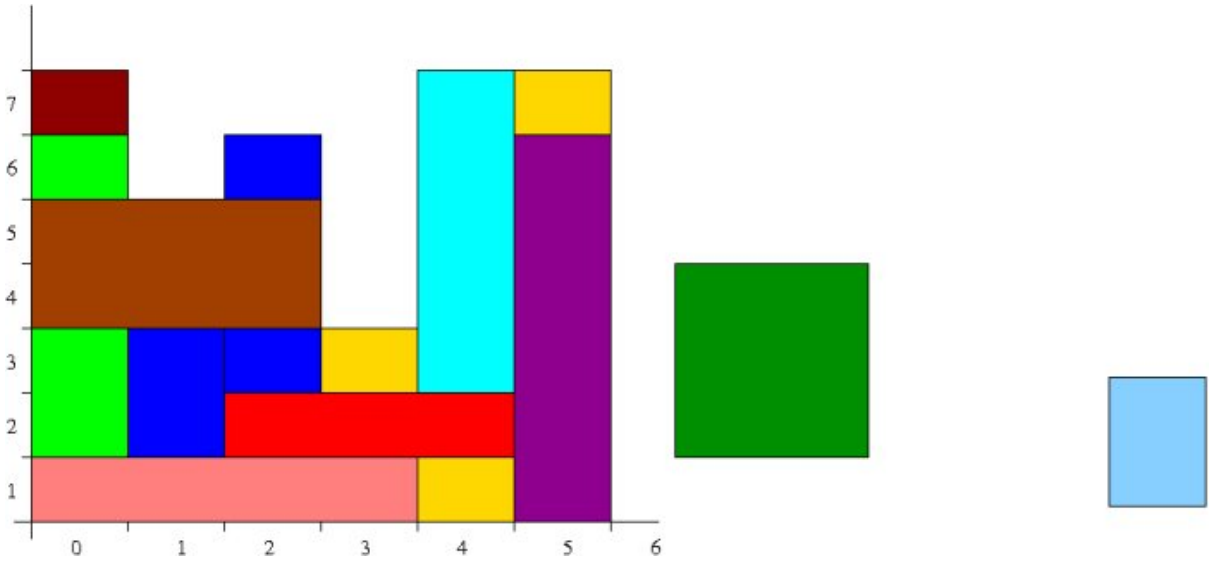

Figure 5.2: Cumulative profile of a solution.

All attributes *sid, x[0],x[1],...,x[k-1], start, duration, end* correspond to domain variables. Typical constraints from the list of constraints C are for instance the fact that a given subset of objects from O do not pairwise overlap. Constraints of the list of constraints C have always two first arguments A_i and O_i (followed by possibly some additional arguments) which respectively specify:

- A list of dimensions (integers between 0 and k-1), or attributes of the objects of O the constraint considers.
- A list of identifiers of the objects to which the constraint applies.

5.2.1 Example and way to implement it

We will explain how to use *geost* although a 2D example. Consider we have 3 objects o_0, o_1, o_2 to place them inside a box B (3x4) such that they don't overlap (see Figure bellow). The first object o_0 has two potential shapes while o_1 and o_2 have one shape. Given that the placement of the objects should be totally inside B this means that the domain of the origins of objects are as follows (we start from 0 this means that the placement space is from 0 to 2 on x and from 0 to 3 on y:

- o_0: x in 0..1, y in 0..1,
- o_1: x in 0..1, y in 0..1,
- o_2: x in 0..1, y in 0..3.

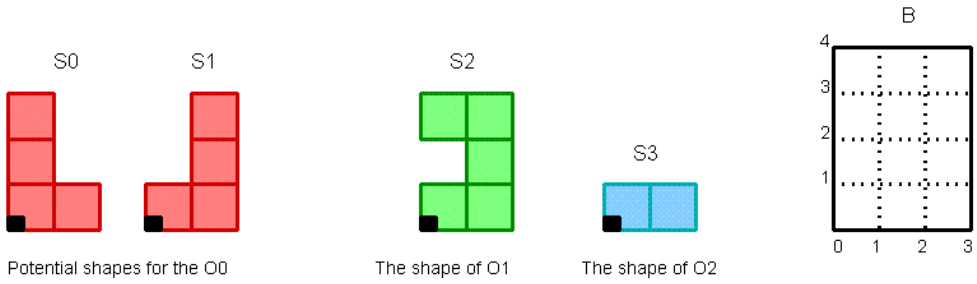

Figure 5.3: Geost objects and shapes

We describe now how to solve this problem by using Choco.

Build a CP model.

To begin the implementation we build a CP model:

```
Model m = new CPModel();
```

Set the Dimension.

Then we first need to specify the dimension k we are working in. This is done by assigning the dimension to a local variable that we will use later:

```
int dim = 2;
```

Create the Objects.

Then we start by creating the objects and store them in a vector as such:

```
Vector<GeostObject> objects = new Vector<GeostObject>();
```

Now we create the first object o_0 by creating all its attributes.

```
int objectId = 0; // object id
IntegerVariable shapeId = Choco.makeIntVar("sid", 0, 1); // shape id (2 possible values)
IntegerVariable coords[] = new IntegerVariable[dim]; // coordinates of the origin
coords[0] = Choco.makeIntVar("x", 0, 1);
coords[1] = Choco.makeIntVar("y", 0, 1);
```

We need to specify 3 more Integer Domain Variables representing the temporal attributes (start, duration and end), which for the current implementation of `geost` are not working, however we need to give them dummy values.

```
IntegerVariable start = Choco.makeIntVar("start", 0, 0);
IntegerVariable duration = Choco.makeIntVar("duration", 1, 1);
IntegerVariable end = Choco.makeIntVar("end", 1, 1);
```

Finally we are ready to add the object 0 to our *objects* Vector:

```
objects.add(new GeostObject(dim, objectId, shapeId, coords, start, duration, end));
```

Now we do the same for the other object o_1 and o_2 and add them to our *objects* vector.

Create the Shifted Boxes.

To create the shapes and their shifted boxes we create the shifted boxes and associate them with the corresponding shapeId. This is done as follows, first we create a Vector called *sb* for example

```
Vector<ShiftedBox> sb = new Vector<ShiftedBox> ();
```

To create the shifted boxes for the shape 0 (that corresponds to o_0), we start by the first shifted box by creating the sid and 2 arrays one to specify the offset of the box in each dimension and one for the size of the box in each dimension:

```
int sid = 0;
int[] offset = {0,0};
int[] sizes = {1,3};
```

Now we add our shiftedbox to the *sb* Vector:

```
sb.add(new ShiftedBox(sid, offset, sizes));
```

We do the same with second shifted box:

```
sb.add(new ShiftedBox(0, new int[]{0,0}, new int[]{2,1}));
```

By the same way we create the shifted boxes corresponding to second shape S_1:

```
sb.add(new ShiftedBox(1, new int[]{0,0}, new int[]{2,1}));
sb.add(new ShiftedBox(1, new int[]{1,0}, new int[]{1,3}));
```

and the third shape S_2 consisting of three shifted boxes:

```
sb.add(new ShiftedBox(2, new int[]{0,0}, new int[]{2,1})) ;
sb.add(new ShiftedBox(2, new int[]{1,0}, new int[]{1,3}));
sb.add(new ShiftedBox(2, new int[]{0,2}, new int[]{2,1}));
```

and finally the last shape S_3

```
sb.add(new ShiftedBox(3, new int[]{0,0}, new int[]{2,1}));
```

Create the constraints.

First we create a Vector called ectr that will contain the external constraints.

```
Vector <ExternalConstraint> ectr = new Vector <ExternalConstraint>();
```

In order to create the non-overlapping constraint we first create an array containing all the dimensions the constraint will be active in (in our example it is all dimensions) and lets name this array *ectrDim* and a list of objects *objOfEctr* that this constraint will apply to (in our example it is all objects).

> Note that in the current implementation of `geost` only the non-overlapping constraint is available. Moreover, *ectrDim* should contain all dimensions and *objOfEctr* should contain all the objects, i.e. the non-overlapping constraint applies to all the objects in all dimensions.

After that we add the constraint to a vector *ectr* that contains all the constraints we want to add. The code for these steps is as follows:

```
int[] ectrDim = new int[dim];
for(i = 0; i < dim; i ++)
    ectrDim[i] = i;
int[] objOfEctr = new int[3];
for(i = 0; i < 3; i ++)
    objOfEctr[i] = objects.elementAt(i).getObjectId();
```

All we need to do now is create the non-overlapping constraint and add it to the *ectr* vector that holds all the constraints. this is done as follows:

```
//Constants.NON_OVERLAPPING indicates the id of the non-overlapping constraint
NonOverlapping n = new NonOverlapping(Constants.NON_OVERLAPPING, ectrDim, objOfEctr);
ectr.add(n);
```

Create the `geost` constraint and add it to the model.

```
Constraint geost = Choco.geost(dim, objects, sb, ectr);
m.addConstraint(geost);
```

Solve the problem.

```
Solver s = new CPSolver();
s.read(m);
s.solve();
```

The full java code can be found here: geostexp.java

5.2.2 Support for Greedy Assignment within the geost Kernel

Motivation and functionality description.

Since, for performance reasons, the `geost` kernel offers a mode where he tries to fix all objects during one single propagation step, we provide a way to specify a preferred order on how to fix all the objects in one single propagation step. This is achieved by:

- Fixing the objects according to the order they were passed to the `geost` kernel.
- When considering one object, fixing its shape variable as well as its coordinates:
  - According to an order on these variables that can be explicitly specified.
  - A value to assign that can either be the smallest or the largest value, also specified by the user.

> Note that the use of the greedy mode assumes that no other constraint is present in the problem.

This is encoded by a term that has exactly the same structure as the term associated to an object of `geost`. The only difference consists of the fact that a variable is replaced by an expression _ (*The character _ denotes the fact that the corresponding attribute is irrelevant, since for instance, we know that it is always fixed*), $\min(I)$ (respectively, $\max(I)$), where I is a strictly positive integer. The meaning is that the corresponding variable should be fixed to its minimum (respectively maximum value) in the order I. We can in fact give a list of vectors $v_1, v_2, \ldots, v_p$ in order to specify how to fix objects $o_{(1+pa)}, o_{(2+pa)}, ..., o_{(p+pa)}$.

This is illustrated by Figure bellow: for instance, Part(I) specifies that we alternatively:

- fix the shape variable of an object to its maximum value (i.e., by using max(1)), fix the x-coordinate of an object to its its minimum value (i.e., by using min(2)), fix the y-coordinate of an object to its its minimum value (i.e., by using min(3)) and
- fix the shape variable of an object to its maximum value (i.e., by using max(1)), fix the x-coordinate of an object to its its maximum value (i.e., by using max(2)), fix the y-coordinate of an object to its its maximum value (i.e., by using max(3)).

In the example associated with Part (I) we successively fix objects $o_1, o_2, o_3, o_4, o_5, o_6$ by alternatively using strategies (1) `object(_,max(1),x[min(2),min(3)])` and (2) `object(_,max(1),x[max(2),max(3))`.

Implementation.

The greedy algorithm for fixing an object o is controlled by a vector v of length $k+1$ such that:

- The shape variable $o.sid$ should be set to its minimum possible value if $v[0] < 0$, and to its maximum possible value otherwise.
- abs($v[1]$)-2 is the most significant dimension (the one that varies the slowest) during the sweep. The values are tried in ascending order if $v[1] < 0$, and in descending order otherwise.
- abs($v[2]$)-2 is the next most significant dimension, and its sign indicates the value order, and so on.

For example, a term `object(_,min(1),[max(3),min(4),max(2)])` is encoded as the vector $[-1, 4, 2, -3]$.

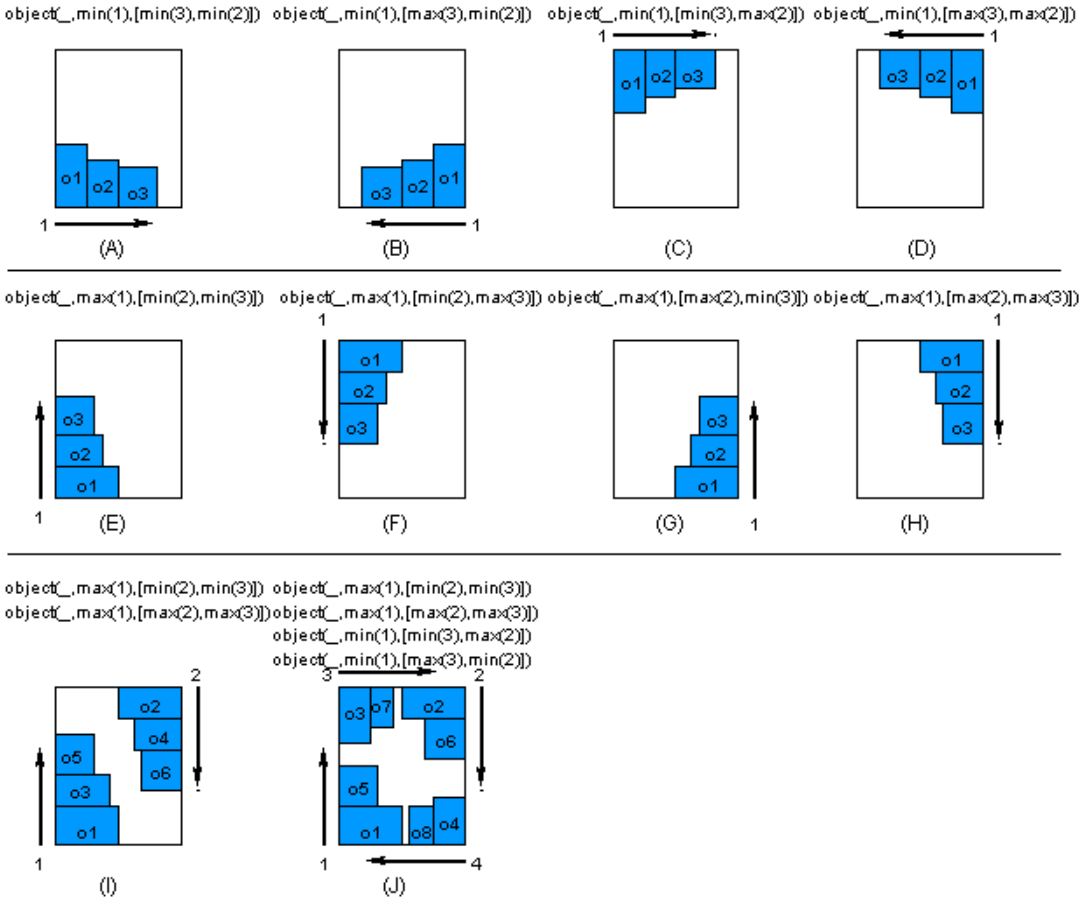

Figure 5.4: Greedy placement

Second example.

We will explain although a 2D example how to take into account of greedy mode. Consider we have 12 identical objects $o_0, o_1, \ldots, o_{11}$ having 4 potential shapes and we want to place them in a box B (7x6) (see Figure bellow). Given that the placement of the objects should be totally inside B this means that the domain of the origins of objects are as follows $x \in [0, 5]$, $y \in [0, 4]$. Moreover, suppose that we want use two strategies when greedy algorithm is called: the term `object(_,min(1),[min(2),min(3)])` for objects $o_0, o_2, o_4, o_6, o_8, o_{10}$, and the term `object(_,max(1),[max(2),max(3)])` for objects $o_1, o_3, o_5, o_7, o_9, o_{11}$. These strategies are encoded respectively as $[-1, -2, -3]$ and $[1, 2, 3]$.

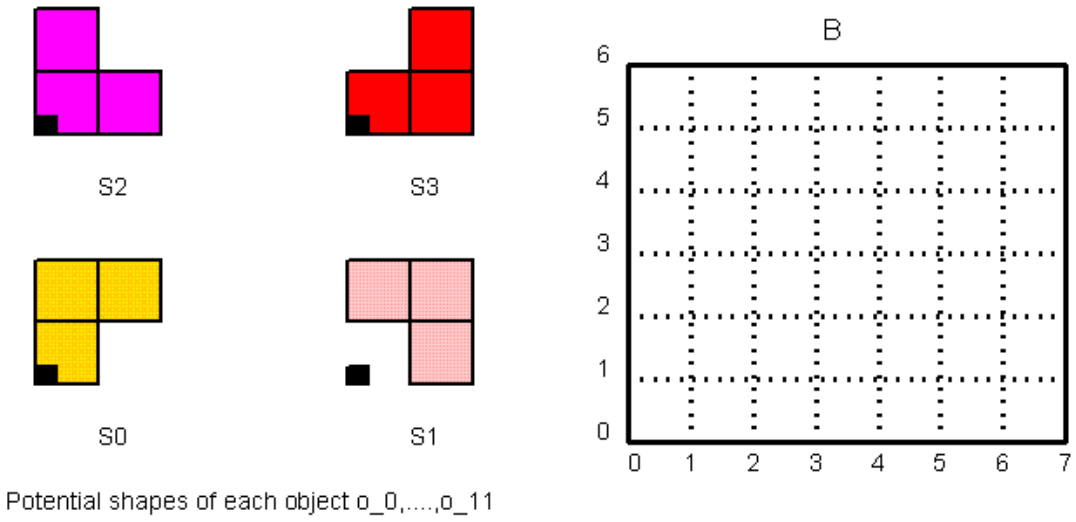

Figure 5.5: A second Geost instance.

We comment only the additional step w.r.t. the preceding example. In fact we just need to create the list of controlling vectors before creating the geost constraint. Each controlling vector is an array:

```
Vector <int[]> ctrlVs = new Vector<int[]>();
int[] v0 = {-1, -2, -3};
int[] v1 = {1, 2, 3};
ctrlVs.add(v0);
ctrlVs.add(v1);
```

and then create the `geost` constraint, by adding the list of controlling vectots as an another argument, as follows:

```
Constraint geost = Choco.geost(dim, objects, sb, ectr,ctrlVs);
m.addConstraint(geost);
```

The full java code can be reached here.

```
import java.util.Vector;

import choco.Choco;
import choco.cp.model.CPModel;
import choco.cp.solver.CPSolver;
import choco.cp.solver.constraints.global.geost.Constants;
import choco.cp.solver.constraints.global.geost.externalConstraints.ExternalConstraint;
import choco.cp.solver.constraints.global.geost.externalConstraints.NonOverlapping;
import choco.cp.solver.constraints.global.geost.geometricPrim.ShiftedBox;
import choco.kernel.model.Model;
import choco.kernel.model.constraints.Constraint;
```

```
import choco.kernel.model.variables.geost.GeostObject;
import choco.kernel.model.variables.integer.IntegerVariable;
import choco.kernel.solver.Solver;

public class GreedyExp {

   // The data
   public static int dim = 2;

   public static int[] domOrigins = { 0, 5, 0, 4 };

   public static int[][] shBoxes = { { 0, 0, 0, 1, 2 }, { 0, 0, 1, 2, 1 },
         { 1, 1, 0, 1, 2 }, { 1, 0, 1, 2, 1 }, { 2, 0, 0, 2, 1 },
         { 2, 0, 0, 1, 2 }, { 3, 0, 0, 2, 1 }, { 3, 1, 0, 1, 2 } };

   public static int[] v0 = { -1, -2, -3 };

   public static int[] v1 = { 1, 2, 3 };

   public static void exp2D() {

      int nbOfObj = 12;

      // create the choco problem
      Model m = new CPModel();

      // Create Objects
      Vector<GeostObject> objects = new Vector<GeostObject>();

      for (int i = 0; i < nbOfObj; i++) {
         IntegerVariable shapeId = Choco.makeIntVar("sid_" + i, 0, 3);
         IntegerVariable coords[] = new IntegerVariable[dim];
         coords[0] = Choco
               .makeIntVar("x_" + i, domOrigins[0], domOrigins[1]);
         coords[1] = Choco
               .makeIntVar("y_" + i, domOrigins[2], domOrigins[3]);

         IntegerVariable start = Choco.makeIntVar("start", 0, 0);
         IntegerVariable duration = Choco.makeIntVar("duration", 1, 1);
         IntegerVariable end = Choco.makeIntVar("end", 1, 1);
         objects.add(new GeostObject(dim, i, shapeId, coords, start,
               duration, end));

      }

      // create shiftedboxes and add them to corresponding shapes
      Vector<ShiftedBox> sb = new Vector<ShiftedBox>();
      for (int i = 0; i < shBoxes.length; i++) {
         int[] offset = { shBoxes[i][1], shBoxes[i][2] };
         int[] sizes = { shBoxes[i][3], shBoxes[i][4] };
         sb.add(new ShiftedBox(shBoxes[i][0], offset, sizes));
      }

      // Create the external constraints vecotr
      Vector<ExternalConstraint> ectr = new Vector<ExternalConstraint>();
      // create the list of dimensions for the external constraint
      int[] ectrDim = new int[dim];
      for (int d = 0; d < dim; d++)
         ectrDim[d] = d;

      // create the list of object ids for the external constraint
```

```
      int[] objOfEctr = new int[nbOfObj];
      for (int j = 0; j < nbOfObj; j++) {
         objOfEctr[j] = objects.elementAt(j).getObjectId();
      }

      // create the non overlapping constraint
      NonOverlapping n = new NonOverlapping(Constants.NON_OVERLAPPING,
            ectrDim, objOfEctr);

      // add the non overlapping constraint to the ectr vector
      ectr.add(n);

      // create the list of controlling vectors
      Vector<int[]> ctrlVs = new Vector<int[]>();
      ctrlVs.add(v0);
      ctrlVs.add(v1);

      // create the geost constraint
      Constraint geost = Choco.geost(dim, objects, sb, ectr, ctrlVs);

      // post the geost constraint to the choco problem
      m.addConstraint(geost);

      // build the solver
      Solver s = new CPSolver();

      // read the problem
      s.read(m);

      // solve the probem
      s.solve();

      // print the solution
      System.out.println(s.pretty());

   }

   public static void main(String args[]) {
      exp2D();
   }
}
```

The placement obtained using the preceding strategies is displayed in the following figure (right side).

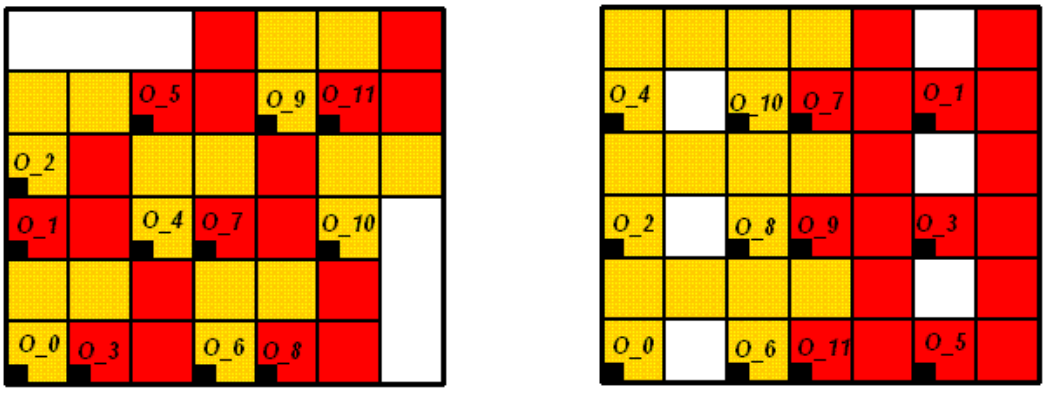

Figure 5.6: A solution placement

Chapter 6

Frequently Asked Questions

6.1 Where can I find Choco ?

See the download page if you want to download a version of Choco library.

6.2 What is the required Java version to run Choco ?

Choco requires java6.

> If you are working on Mac OS X 10.4 Tiger or if you do not have an Intel processor, you probably can not install java 6 on your OS. Please, take a look at Soy latte, which goals are "support for Java 6 Development on Mac OS X 10.4 and 10.5, OpenJDK support for Java 7 on Mac OS X and On-time release of Java 7 for Mac OS X".

6.3 How to add the Choco library to my project?

You just need to add Choco.X.x.x.jar to your classpath.
IntelliJ:

- Go to "File/Settings"
- Select "Project Settings"
- Click on "Librairies" then on [+]
- Enter "Choco" as the library name, and OK
- Choose your project, and OK,
- Click on "Attach Jar Directories" and choose the directory where you put the Choco jar.

fix hyperlinkflowplay¿videos:intellij.flvHow to add choco to IntelliJ in video
Eclipse:

- Go to "Project/Properties",
- Select "Java Build Path" on the left menu
- On the right, select "Librairies"
- Click on "Add Externals JARs..." button

- Select the Choco jar file

fix hyperlinkflowplay¿videos:eclipse.flvHow to add choco to Eclipse in video

> If you work with **the source and not the jar of Choco**, do not forget to also add the automaton.jar and junit.jar, available in the lib/ directory

6.4 Why can't I see the Choco API?

To have the Choco API available, you must make the following import in your class file:

```
import static choco.Choco.*;
```

6.5 How to know the value of my variable in the Solver ?

There are two different kinds of variables: those associated with the Model (like `IntegerVariable`, `SetVariable`,...) and those associated with the Solver (like `IntDomainVar`, `SetVar`,...). The second type is a Solver interpretation of the first one (which is only declarative). After having defined your model with variables and constraints, it has to be read by the Solver. After that, a Solver object is created. You can access the Variable Model *value* through the Solver using the following method of the Solver: `solver.getVar(Variable v);` where v is a Model variable (or an array of Model variables) and it returns a Solver variable.

6.6 How do I use constant value inside constraint ?

Some constraints doesn't provide API with java object (like **int**, **double** or **Integer**). You can define *constant* variable (ie, variable with one unique value) liek this:

```
IntegerVariable one = constant(1);
RealVariable one = constant(1.0);
```

And, you can use this *variable* inside the constraint:

```
Model m = new CPModel();

IntegerVariable x = makeIntVar("x", 0, 10);
IntegerVariable two = constant(2);
IntegerVariable maximum = makeIntVar("max", 0, 15);

m.addConstraint(eq(maximum, max(x, two));
```

Do not forget that some contraints provide api with java object.

6.7 How can I use Choco to solve CSP'08 benchmark ?

You can easily load an XML file of the CSP'08 competition and solve it with Choco. To load the file, we use the XMLParser available here:

```
String fileName = "../../ProblemsData/CSPCompet/intension/nonregres/graph1.xml";
File instance = new File(fileName);
XmlModel xs = new XmlModel(); // a class to ease loading and solving CSP'08 xml file
InstanceParser parser = xs.load(instance); // loading of the CPS'08 xml file
```

Once the file has been loaded, a Model object is build from the InstanceParser object:

```
CPModel model = xs.buildModel(parser); // Creation of the model
```

At this point, you can choose to solve this model with a pre-processing step. The pre-processing step analyzes variables and constraints, makes some specific choices to improve the resolution. Concerning variables, it analyzes domains and constraints and choose what seems to be the best kind of domain (for example, enumerated or bounded domain), or add one variable where large number of variables are equals, ... Concerning constraints, it detects clique of differences or disjunctions and state the corresponding global constraints, breaks symetries, detects distance... Then, it can also choose the search strategy. To do this, use the following code:

```
PreProcessCPSolver s = xs.solve(model); // Build a BlackBoxSolver and solve it.
```

Finally, you can print informations concerning the resolution:

```
\lstinline|xs.postAnalyze(instance, parser, s);
```

You can easily solve benchmarks of CSP'08 competition, or with your own problem modelize in CSP'08 xml format.

6.8 How do I use the build.xml file ?

The choco project provides an ant script `build.xml` for the most usual tasks of the project. In this section, we show how to run these tasks with a terminal. Ant is fully integrated in most of Java IDE but we will not talk about it.

First, we are going into the root directory of choco

```
nono@arrakis:~\$ cd /path/to/choco/
nono@arrakis:~/workspace/Choco-2.0\$ ls
bin build.xml checkstyle.xml Choco2.0.0.iml choco-ruleset.xml dev lib pom.xml
```

Then, try a simple

```
nono@arrakis:~/workspace/Choco-2.0\$ ant
Buildfile: build.xml

init:
     [echo] Ant version : Apache Ant version 1.7.0 compiled on August 29 2007
     [echo] Java version : 1.6.0_06
     [echo] build of project JChoco : June 30 2008

help:
     [echo] be careful, there could have a bug in eclipse with this help message.
     [echo] In this case, type "ant␣-p␣popart/build.xml" in a terminal.
     [exec] Buildfile: build.xml
     [exec]
     [exec] Main targets:
     [exec]
     [exec] clean --> deletes everything that seems useless
     [exec] compile --> compiles everything
     [exec] dist --> makes the distribution package (jar, src.zip, doc.zip)
     [exec] doc --> generates the javadoc
     [exec] exec-junit-test --> executes all junit tests.
     [exec] exec-pmd --> analyzes code with PMD and CPD
     [exec] help --> print this help
     [exec] Default target: help

BUILD SUCCESSFUL
Total time: 1 second
```

Most of these tasks do not have some special requirements. However, you could need specific settings

to run `exec-junit-test` and `exec-pmd`. You need to have `junit` and `ant-junit` jars in your classpath to run `exec_junit-test`. But, it works for my first attempt without any changes. You need to set `pmd` jar in your classpath and probably to update the property `pmd.xslt` to run `exec_junit-test`. Supposes that you have installed `pmd` in `/path/to/pmd`. You have to reset the location of the following property:

```
<property name="pmd.xslt" location="/path/to/pmd/etc/xslt/wz-pmd-report.xslt" />
```

Finally you can run the task with :

```
ant -lib /path/to/pmd/lib/pmd-4.1.jar exec-pmd
```

It seems that using PMD with your IDE is more effective. The integration offers many filtering options and allows to correct your code on the fly.

A post is opened for feedbacks, for new feature requests, and for any comments about the `build.xml` file

6.9 Why do I have a error when I add my constraint ?

If you have a error message like this:

Component class could not be found: my.package.and.my.Constraint.ConstraintManager

and if the ConstraintManager is an inner class of your constraint, you must define the name in the component name like this:

```
my.package.and.my.Constraint\$ConstraintManager
```

For more details, see define your own constraint.

6.10 How do I upgrade my program to Choco2.0 ?

Without being very precise (see the documentation if you want more details), it is really easy to transpose a program implemented on an old version of Choco to Choco2.0.

No Problem!! The `Problem` class does not exist anymore. It has been replaced by two new classes: `CPModel` and `CPSolver` that implement the interfaces `Model` and `Solver`. The model allows you to declare your variables and constraints and the solver allows you to define some search strategies and solve your model. As different kinds of Model and Solver will be available, everything concerning Variables and Constraints is included in the new class `Choco`.

Now, let us see in a few steps how to transpose your program. Consider that you created the following program:

```
// Creation of the problem
Problem pb = new Problem();

// Declaration of variables
IntDomainVar v1 = pb.makeEnumIntVar("v1", 1, 10);
IntDomainVar v2 = pb.makeEnumIntVar("v1", 1, 10);

// Declaration of constraints
Constraint c1 = pb.neq(v1,v2);
pb.post(c1);
// Declaration of a user constraint
Constraint prime-number = MyConstraint(v1, v2);
pb.post(prime-number);

// Definition of a search strategy
pb.getSolver().setVarSelector(new StaticVarOrder(v1, v2));
pb.getSolver().setValIterator(new IncreasingDomain());

// Resolution of the problem
pb.solve();
```

```
// Print the solution
System.out.println("v1"+v1.getVal());
System.out.println("v2"+v2.getVal());
```

- **A Problem becomes a Model and a Solver**

```
// Creation of the problem
Problem pb = new Problem();
```

becomes

```
// Creation of the Model
Model m = new CPModel();
//Creation of the Solver
Solver s = new CPSolver();
```

- **Variables are independent of a Problem or a Model**

```
// Declaration of variables
IntDomainVar v1 = pb.makeEnumIntVar("v1", 1, 10);
IntDomainVar v2 = pb.makeEnumIntVar("v1", 1, 10);
```

becomes

```
// add import:
import static choco.Choco.*;
//...
// Declaration of variables
IntegerVariable v1 = makeIntVar("v1", 1, 10);
IntegerVariable v2 = makeIntVar("v1", 1, 10);
m.addVariable("cp:enum", v1, v2);
```

- **Easy declaration of constraints**

```
// Declaration of constraints
Constraint c1 = pb.neq(v1,v2);
pb.post(c1);
```

becomes

```
// add import (same than Variables):
import static choco.Choco.*;
//...
// Declaration of constraints
Constraint c1 = neq(v1,v2);
m.addConstraint(c1);
```

- **A specific way to define user constraints**

```
// Declaration of a user constraint
Constraint prime-number = myConstraint(v1, v2);
pb.post(prime-number);
```

becomes

```
// Declaration of a user constraint
m.addConstraint(new ComponentConstraint(MyManager.class, null, v1, v2));
```

- **Do not forget to read the model**

It is a ***new step***, it has to be done!

```
// Read the model
s.read(m);
```

- **Clear definition of the search strategy**

```
// Definition of a search strategy
pb.getSolver().setVarSelector(new StaticVarOrder(v1, v2));
pb.getSolver().setValIterator(new IncreasingDomain());
```

becomes

```
// Definition of a search strategy
s.setVarIntSelector(new StaticVarOrder(s.getVar(v1, v2)));
s.setValIntIterator(new IncreasingDomain());
```

- **And the resolution**

```
// Resolution of the problem
pb.solve();
```

becomes

```
// Resolution of the model
s.solve();
```

- **Printing the solution**

```
// Print the solution
System.out.println("v1"+v1.getVal());
System.out.println("v2"+v2.getVal());
```

becomes

```
// Print the solution
System.out.println("v1"+s.getVar(v1).getVal());
System.out.println("v2"+s.getVar(v2).getVal());
```

And it's done! We obtain the following code:

```
import static choco.Choco.*;
...
// Creation of the Model
Model m = new CPModel();
//Creation of the Solver
Solver s = new CPSolver();

// Declaration of variables
IntegerVariable v1 = makeIntVar("v1", 1, 10);
IntegerVariable v2 = makeIntVar("v2", 1, 10);
m.addVariable("cp:enum",v1, v2);

// Declaration of constraints
Constraint c1 = neq(v1,v2);
m.addConstraint(c1);
// Declaration of a user constraint
m.addConstraint(new ComponentConstraint(MyManager.class, null, v1, v2));
```

```
// Read the model
s.read(m);

// Definition of a search strategy
s.setVarIntSelector(new StaticVarOrder(s.getVar(v1, v2)));
s.setValIntIterator(new IncreasingDomain());

// Resolution of the model
s.solve();

// Print the solution
System.out.println("v1"+s.getVar(v1).getVal());
System.out.println("v2"+s.getVar(v2).getVal());
```

6.11 Are bounds with positive and negative infinity supported within Choco?

Integer or Double infinity bounds are not really appreciate by CHOCO :) Because, during propagation, a basic test is done on bounds and the following operation can be applied: *upper bound +1*. As `Integer.MAX_VALUE+1` is equal to `Integer.MIN_VALUE`, it can corrupt the propagation.

If you really want to have a large domain, a division with 10 should be sufficient:

```
IntegerVariable v1 = makeIntVar("v1", Integer.MIN_VALUE/10, Integer.MIN_VALUE/10);
RealVariable a1 = makeRealVar("A1", Double.NEGATIVE_INFINITY/10, Double.POSITIVE_INFINITY/10);
```

Part II

Elements of Choco

Chapter 7

Variables (Model)

This section describes the three kinds of variables that can be used within a Choco Model.

7.1 Integer variables

`IntegerVariable` is a variable whose associated domain is made of integer values.

constructors:

Choco method	return type
`makeIntVar(String name, int lowB, int uppB, String... options)`	`IntegerVariable`
`makeIntVar(String name, List<Integer> values, String... options)`	`IntegerVariable`
`makeIntVar(String name, int[] values, String... options)`	`IntegerVariable`
`makeBooleanVar(String name, String... options)`	`IntegerVariable`
`makeIntVarArray(String name, int dim, int lowB, int uppB, String... options)`	`IntegerVariable[]`
`makeIntVarArray(String name, int dim, int[] values, String... options)`	`IntegerVariable[]`
`makeBooleanVarArray(String name, int dim, String... options)`	`IntegerVariable[]`
`makeIntVarArray(String name, int dim1, int dim2, int lowB, int uppB, String ... options)`	`IntegerVariable[][]`
`makeIntVarArray(String name, int dim1, int dim2, int[] values, String... options)`	`IntegerVariable[][]`

options:

- *no option* : equivalent to option `cp:enum`
- `cp:enum` : to force Solver to create enumerated domain for the variable. It is a domain in which holes can be created by the solver. It should be used when discrete and quite small domains are needed and when constraints performing Arc Consistency are added on the corresponding variables. Implemented by a `BitSet` object.
- `cp:bound` : to force Solver to create bounded domain for the variable. It is a domain where only bound propagation can be done (no holes). It is very well suited when constraints performing only Bound Consistency are added on the corresponding variables. It must be used when large domains are needed. Implemented by two integers.
- `cp:binary` : to force Solver to create binary domain for the variable. It is a [0,1] domain. Some operations are improved and leads to an direct instantiation, if necessary. It must be used with two size domain. Implemented by a shared `BitSet` object, common to each 32 binary variables (to deal with 64 bits bitset).
- `cp:link` : to force Solver to create linked list domain for the variable. It is an enumerated domain where holes can be done and every values has a link to the previous value and to the next value. It is built by giving its name and its bounds: lower bound and upper bound. It must be used when

the very small domains are needed, because although linked list domain consumes more memory than the `BitSet` implementation, it can provide good performance as iteration over the domain is made in constant time. Implemented by a `LinkedList` object.

- `cp:btree` : to force Solver to create binary tree domain for the variable. *Under development.*
- `cp:blist` : to force Solver to create bipartite list domain for the variable. It is a domain where unavailable values are placed in the left part of the list, the other one on the right one.
- `cp:decision` : to force variable to be a decisional one
- `cp:no_decision` : to force variable to be removed from the pool of decisional variables
- `cp:objective` : to define the variable to be the one to optimize

methods:

- `removeVal(int val)`: remove value *val* from the domain of the current variable

A variable with $\{0, 1\}$ domain is automatically considered as boolean domain.

Example:

```
IntegerVariable ivar1 = makeIntVar("ivar1", -10, 10);
IntegerVariable ivar2 = makeIntVar("ivar2", 0, 10000, "cp:bound", "cp:decision");
IntegerVariable bool = makeIntVar("bool", 0, 1, "cp:binary");
```

7.2 Real variables

`RealVariable` is a variable whose associated domain is made of real values. Only enumerated domain is available for real variables.

Such domain are memory consuming. In order to minimize the memory use and to have the precision you need, the model offers a way to set a precision (default value is 1.0e-6):

```
Model m = new CPModel();
m.setPrecision(0.01);
```

constructor:

Choco method	return type
`makeRealVar(String name, double lowB, double uppB, String... options)`	`RealVariable`

options:

- *no option* : no particular choice on decision or objective.
- `cp:decision` : to force variable to be a decisional one
- `cp:no_decision` : to force variable to be removed from the pool of decisionnal variables
- `cp:objective` : to define the variable to be the one to optimize

Example:

```
RealVariable rvar1 = makeRealVar("rvar0", -10.0, 10.0);
RealVariable rvar2 = makeRealVar("rvar2", 0.0, 100.0,"cp:decision", "cp:objective");
```

7.3 Set variables

`SetVariable` is high level modeling tool. It allows to represent variable whose values are sets. A SetVariable on integer values between $[1, n]$ has $2*n$ values (every possible subsets of $\{1..n\}$). This makes an exponential number of values and the domain is represented with two bounds corresponding to the intersection of all possible sets (called the kernel) and the union of all possible sets (called the envelope) which are the possible candidate values for the variable. The consistency achieved on SetVariables is therefore a kind of bound consistency.

constructors:

Choco method	return type
`makeSetVar(String name, int lowB, int uppB, String... options)`	`SetVariable`
`makeSetVarArray(String name, int dim, int lowB, int uppB, String... options)`	`SetVariable[]`

options:

- *no option* : equivalent to option `cp:enum`
- `cp:enum` : to force Solver to create `SetVariable` with enumerated domain for the caridinality variable. It is a domain in which holes can be created by the solver. It should be used when set variable cardinality domain is discrete, quite small and constraints performing reasonings on holes in the cardinality are present in the model. Implemeted by two `BitSets` for upper and lower bounds and an enumerated `IntegerVariable` for the cardinality.
- `cp:bound` : to force Solver to create `SetVariable` with bounded cardinality. It is a domain where only bound propagation can be done (no holes). It is very well suited when constraints performing only Bound Consistency are added on the corresponding variables. It must be used when large domains are needed. Implemented by two integers.
- `cp:decision` : to force variable to be a decisional one
- `cp:no_decision` : to force variable to be removed from the pool of decisionnal variables
- `cp:objective` : to define the variable to be the one to optimize

The variable representing the cardinality can be accessed and constrained using method `getCard()` that returns an `IntegerVariable` object.

Example:

```
SetVariable svar1 = makeSetVar("svar1", -10, 10);
setVariable svar2 = makeSetVar("svar2", 0, 10000, "cp:bound", "cp:no_decision");
```

Set variables are illustrated on the ternary Steiner problem.

Chapter 8

Operators

This section lists and details the operators that can be used within a Choco Model to combine variables in expressions.

8.1 abs (operator)

Returns an expression variable that represents the absolute value of the argument ($|n|$).

- **API** : abs(IntegerExpressionVariable n)
- **return type** : IntegerExpressionVariable
- **options** : *n/a*
- **favorite domain** : unknown

Example:

```
Model m = new CPModel();
IntegerVariable x = makeIntVar("x", 1, 5, "cp:enum");
IntegerVariable y = makeIntVar("y", -5, 5, "cp:enum");
m.addConstraint(eq(abs(x), y));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

8.2 cos (operator)

Returns an expression variable corresponding to the cosinus value of the argument ($cos(x)$).

- **API** : cos(RealExpressionVariable exp)
- **return type** : RealExpressionVariable
- **options** : *n/a*
- **favorite domain** : real

Example: *No valid example for the moment*

8.3 div (operator)

Returns an expression variable that represents the *integerquotient* of the division of the first argument variable by the second one (n_1/n_2).

- **API** :
  - div(IntegerExpressionVariable n1, IntegerExpressionVariable n2)
  - div(IntegerExpressionVariable n1, int n2)
  - div(int n1, IntegerExpressionVariable n2)
- **return type** : IntegerExpressionVariable
- **options** : *n/a*
- **favorite domain** : *n/a*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 10);
IntegerVariable w = makeIntVar("w", 22, 44);
IntegerVariable z = makeIntVar("z", 12, 21);
m.addConstraint(eq(z, div(w, x)));
s.read(m);
s.solve();
```

8.4 FALSE (operator)

Returns an expression always equals to *false*.

8.5 ifThenElse (operator)

To complete

8.6 max (operator)

Returns an expression variable equals to the greater value of the argument ($max(x_1, x_2, ..., x_n)$).

- **API** :
  - max(IntegerExpressionVariable x1, IntegerExpressionVariable x2)
  - max(int x1, IntegerExpressionVariable x2)
  - max(IntegerExpressionVariable x1, int x2)
  - max(IntegerExpressionVariable[] x)
- **return type**: IntegerExpressionVariable
- **options** : *n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
m.setDefaultExpressionDecomposition(true);
IntegerVariable[] v = makeIntVarArray("v", 3, -3, 3);
IntegerVariable maxv = makeIntVar("max", -3, 3);
Constraint c = eq(maxv, max(v));
m.addConstraint(c);
Solver s = new CPSolver();
s.read(m);
s.solveAll();
```

8.7 min (operator)

Returns an expression variable equals to the smaller value of the argument ($min(x_1, x_2, ..., x_n)$).

- **API** :
  - min(IntegerExpressionVariable x1, IntegerExpressionVariable x2)
  - min(int x1, IntegerExpressionVariable x2)
  - min(IntegerExpressionVariable x1, int x2)
  - min(IntegerExpressionVariable[] x)
- **return type**: IntegerExpressionVariable
- **options** : *n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
m.setDefaultExpressionDecomposition(true);
IntegerVariable[] v = makeIntVarArray("v", 3, -3, 3);
IntegerVariable minv = makeIntVar("min", -3, 3);
Constraint c = eq(minv, min(v));
m.addConstraint(c);
Solver s = new CPSolver();
s.read(m);
s.solveAll();
```

8.8 minus (operator)

Returns an expression variable that corresponding to the difference between the two arguments ($x - y$).

- **API** :
  - `minus(IntegerExpressionVariable x, IntegerExpressionVariable y)`
  - `minus(IntegerExpressionVariable x, int y)`
  - `minus(int x, IntegerExpressionVariable y)`
  - `minus(RealExpressionVariable x, RealExpressionVariable y)`
  - `minus(RealExpressionVariable x, double y)`
  - `minus(double x, RealExpressionVariable y)`
- **return type** :
  - `IntegerExpressionVariable`, if parameters are `IntegerExpressionVariable`

- `RealExpressionVariable`, if parameters are `RealExpressionVariable`

- **options** : *n/a*
- **favorite domain** : *to complete*

Example

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable a = makeIntVar("a", 0, 4);
m.addConstraint(eq(minus(a, 1), 2));
s.read(m);
s.solve();
```

8.9 mod (operator)

Returns an expression variable that represents the integer remainder of the division of the first argument variable by the second one ($x_1 \% x_2$).

- **API**:
  - mod(IntegerExpressionVariable x1, IntegerExpressionVariable x2)
  - mod(int x1, IntegerExpressionVariable x2)
  - mod(IntegerExpressionVariable x1, int x2)
- **return type** : `IntegerExpressionVariable`
- **options** : *n/a*
- **favorite domain** : *n/a*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 10);
IntegerVariable w = makeIntVar("w", 22, 44);
m.addConstraint(eq(1, mod(w, x)));
s.read(m);
s.solve();
```

8.10 mult (operator)

Returns an expression variable that corresponding to the product of variables in argument ($x * y$).

- **API** :
  - mult(IntegerExpressionVariable x, IntegerExpressionVariable y)
  - mult(IntegerExpressionVariable x, int y)
  - mult(int x, IntegerExpressionVariable y)
  - mult(RealExpressionVariable x, RealExpressionVariable y)
  - mult(RealExpressionVariable x, double y)
  - mult(double x, RealExpressionVariable y)
- **return type** :
  - `IntegerExpressionVariable`, if parameters are `IntegerExpressionVariable`

- RealExpressionVariable, if parameters are RealExpressionVariable

- **options** : *n/a*
- **favorite domain** : *to complete*

Example

```
CPModel m = new CPModel();
IntegerVariable x = makeIntVar("x", -10, 10);
IntegerVariable z = makeIntVar("z", -10, 10);
IntegerVariable w = makeIntVar("w", -10, 10);
m.addVariables(x, z, w);
CPSolver s = new CPSolver();
// x >= z * w
Constraint exp = geq(x, mult(z, w));
m.setDefaultExpressionDecomposition(true);
m.addConstraint(exp);
s.read(m);
s.solveAll();
```

8.11 neg (operator)

Returns an expression variable that is the opposite of the expression integer variable in argument $(-x)$.

- **API** : neg(IntegerExpressionVariable x)
- **return type** : IntegerExpressionVariable
- **options** : *n/a*
- **favorite domain** : *n/a*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", -10, 10);
IntegerVariable w = makeIntVar("w", -10, 10);
// -x = w - 20
m.addConstraint(eq(neg(x), minus(w, 20)));
s.read(m);
s.solve();
```

8.12 plus (operator)

Returns an expression variable that corresponding to the sum of the two arguments $(x + y)$.

- **API** :
  - plus(IntegerExpressionVariable x, IntegerExpressionVariable y)
  - plus(IntegerExpressionVariable x, int y)
  - plus(int x, IntegerExpressionVariable y)
  - plus(RealExpressionVariable x, RealExpressionVariable y)
  - plus(RealExpressionVariable x, double y)
  - plus(double x, RealExpressionVariable y)
- **return type** :

- IntegerExpressionVariable, if parameters are IntegerExpressionVariable
- RealExpressionVariable, if parameters are RealExpressionVariable

- **options** : *n/a*
- **favorite domain** : *to complete*

Example

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable a = makeIntVar("a", 0, 4);
// a + 1 = 2
m.addConstraint(eq(plus(a, 1), 2));
s.read(m);
s.solve();
```

8.13 power (operator)

Returns an expression variable that represents the first argument raised to the power of the second argument (x^y).

- **API** :
  - power(IntegerExpressionVariable x, IntegerExpressionVariable y)
  - power(int x, IntegerExpressionVariable y)
  - power(IntegerExpressionVariable x, int y)
  - power(RealExpressionVariable x, int y)
- **return type**:
  - IntegerExpressionVariable, if parameters are IntegerExpressionVariable
  - RealExpressionVariable, if parameters are RealExpressionVariable
- **option** : *n/a*
- **favorite domain** : *to complete*

Example :

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 0, 10);
IntegerVariable y = makeIntVar("y", 2, 4);
IntegerVariable z = makeIntVar("z", 28, 80);
m.addConstraint(eq(z, power(x, y)));
s.read(m);
s.solve();
```

8.14 scalar (operator)

Return an integer expression that corresponds to the scalar product of coefficients array and variables array ($c_1 * x_1 + c_2 * x_2 + ... + c_n * x_n$).

- **API** :
  - scalar(int[] c, IntegerVariable[] x)
  - scalar(IntegerVariable[] x, int[] c)

- **return type** : IntegerExpressionVariable
- **options** : *n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();

IntegerVariable[] vars = makeIntVarArray("C", 9, 1, 10);
int[] coefficients = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9};
m.addConstraint(eq(165, scalar(coefficients, vars)));

s.read(m);
s.solve();
System.out.print("165 = (" + coefficients[0] + "*" + s.getVar(vars[0]).getVal()+")");
for (int i = 1; i < vars.length; i++) {
    System.out.print(" + (" + coefficients[i] + "*" + s.getVar(vars[i]).getVal()+")");
}
System.out.println();
```

8.15 sin (operator)

Returns a real variable that corresponding to the sinus value of the argument ($sin(x)$).

- **API** : sin(RealExpressionVariable exp)
- **return type** : RealExpressionVariable
- **options** : *n/a*
- **favorite domain** : real

Example: *No valid example for the moment*

8.16 sum (operator)

Return an integer expression that corresponds to the sum of the variables given in argument ($x_1 + x_2 + ... + x_n$).

- **API**: sum(IntegerVariable... lv)
- **return type** : IntegerExpressionVariable
- **options** : *n/a*
- **favorite domain** : *to complete*

Example :

```
Model m = new CPModel();
Solver s = new CPSolver();

IntegerVariable[] vars = makeIntVarArray("C", 10, 1, 10);
m.addConstraint(eq(99, sum(vars)));

s.read(m);
s.solve();
if(s.isFeasible()){
```

```
            System.out.print("99␣=␣" + s.getVar(vars[0]).getVal());
            for (int i = 1; i < vars.length; i++) {
                System.out.print("␣+␣"+s.getVar(vars[i]).getVal());
            }
            System.out.println();
        }
```

8.17 TRUE (operator)

Returns an expression always equals to *true*.

Chapter 9

Constraints

This section lists and details the constraints currently available in Choco.

9.1 abs (constraint)

`abs`(x, y) states that x is the absolute value of y:

$$x = |y|$$

- **API** : `abs(IntegerVariable x, IntegerVariable y)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : enumerated

Example:

```
Model m = new CPModel();
IntegerVariable x = makeIntVar("x", 1, 5, "cp:enum");
IntegerVariable y = makeIntVar("y", -5, 5, "cp:enum");
m.addConstraint(abs(x, y));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

9.2 allDifferent (constraint)

`allDifferent`$(x_1, \ldots, x_n)$ states that the arguments have pairwise distinct values:

$$x_i \neq x_j, \quad \forall\, i \neq j$$

This constraint is useful for some matching problems. Notice that the filtering algorithm used will depend on the nature (enumerated or bounded) of the variables: when *enumerated*, the constraint refers to the alldifferent of [**?**]; when *bounded*, a dedicated algorithm for bound propagation is used [**?**].

- **API** :

- allDifferent(IntegerVariable... x)
- allDifferent(String options, IntegerVariable... x)

- **return type** : Constraint
- **options** :
  - *no option* clever choice made on domains of given variables
  - cp:ac for [?] implementation of arc consistency
  - cp:bc for [?] implementation of bound consistency
  - cp:clique for propagating the clique of differences
- **favorite domain** : depending of options.
- **references** :
  - [?]: *A filtering algorithm for constraints of difference in CSPs*
  - [?]: *A fast and simple algorithm for bounds consistency of the alldifferent constraint*
  - global constraint catalog: alldifferent

Example:

```
int n = 8;
CPModel m = new CPModel();
IntegerVariable[] queens = new IntegerVariable[n];
IntegerVariable[] diag1 = new IntegerVariable[n];
IntegerVariable[] diag2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queens[i] = makeIntVar("Q" + i, 1, n);
    diag1[i] = makeIntVar("D1" + i, 1, 2 * n);
    diag2[i] = makeIntVar("D2" + i, -n + 1, n);
}
m.addConstraint(allDifferent(queens));
for (int i = 0; i < n; i++) {
    m.addConstraint(eq(diag1[i], plus(queens[i], i)));
    m.addConstraint(eq(diag2[i], minus(queens[i], i)));
}
m.addConstraint("cp:clique", allDifferent(diag1));
m.addConstraint("cp:clique", allDifferent(diag2));
// diagonal constraints
CPSolver s = new CPSolver();
s.read(m);
long tps = System.currentTimeMillis();
s.solveAll();
System.out.println("tps␣nreines1␣" + (System.currentTimeMillis() - tps) + "␣nbNode␣" + s
    .
       getNodeCount());
```

9.3 and (constraint)

and$(c_1, \ldots, c_n)$ states that every constraints in arguments are satisfied:

$$c_1 \wedge c_2 \wedge \ldots \wedge c_n$$

- **API** : and(Constraint... c)

- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*
- **references** :
  global constraint catalog: and

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 1);
IntegerVariable v2 = makeIntVar("v2", 0, 1);
m.addConstraint(and(eq(v1, 1), eq(v2, 1)));
s.read(m);
s.solve();
```

9.4 atMostNValue (constraint)

`atMostNValue`(x, z) states that the number of different values occurring in the array of variables x is at most z:

$$z \geq |\{x_1, \ldots, x_n\}|$$

- **API** : `atMostNValue(IntegerVariable[] x, IntegerVariable z)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*
- **references** :
  - [**?**] *Filtering algorithms for the NValue constraint*
  - global constraint catalog: atmost_nvalue

Example:

```
Model m = new CPModel();
CPSolver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 1, 1);
IntegerVariable v2 = makeIntVar("v2", 2, 2);
IntegerVariable v3 = makeIntVar("v3", 3, 3);
IntegerVariable v4 = makeIntVar("v4", 3, 4);
IntegerVariable n = makeIntVar("n", 3, 3);
Constraint c = atMostNValue(new IntegerVariable[]{v1, v2, v3, v4}, n);
m.addConstraint(c);
s.read(m);
s.solve();
```

9.5 boolChanneling (constraint)

`boolChanneling`(b, x, v) states that b is true if and only if x is equal to v:

$$b \iff (x = v)$$

It acts as an observer of value v. Imagine a bin packing problem where variable x tells you on which a given bin object is placed. By stating the boolean channeling, b is true if and only if the object is placed on bin v, the knapsack constraint for bin v can then be easily stated as a scalar of the boolean variables.

- **API** : `boolChanneling(IntegerVariable b, IntegerVariable x, int v)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : enumerated for x

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable bool = makeIntVar("bool", 0, 1);
IntegerVariable x = makeIntVar("x", 0, 5);
m.addConstraint(boolChanneling(bool, x, 4));
s.read(m);
s.solveAll();
```

9.6 cumulative (constraint)

`cumulative(start,duration,height,capacity)` states that a set of tasks (defined by their starting times, finishing dates, durations and heights (or consumptions)) are executed on a cumulative resource of limited capacity. That is, the total height of the tasks which are executed at any time t does not exceed the capacity of the resource:

$$\sum_{\{i \mid \texttt{start}[i] \leq t < \texttt{start}[i]+\texttt{duration}[i]\}} \texttt{height}[i] \leq \texttt{capacity}, \quad (\forall \text{ time } t)$$

The notion of task does not exist yet in Choco. The `cumulative` takes therefore as input, several arrays of integer variables (of same size n) denoting the starting, duration, and height of each task. When the array of finishing times is also specified, the constraint ensures that `start[i] + duration[i] = end[i]` for all task i. As usual, a task is executed in the interval `[start,end-1]`.

A tutorial on the use of this constraint is available here

- **API** :
  - `cumulative(IntegerVariable[] start, IntegerVariable[] end, IntegerVariable[] duration, IntegerVariable[] height, IntegerVariable capa, String... options)`
  - `cumulative(IntegerVariable[] start, IntegerVariable[] end, IntegerVariable[] duration, int[] height, int capa, String... options)`

- cumulative(IntegerVariable[] start, IntegerVariable[] duration, IntegerVariable[] height , IntegerVariable capa, String... options)

- **return type** : Constraint
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  - [?] *A new multi-resource cumulatives constraint with negative heights*
  - global constraint catalog: cumulative

Example:

```
CPModel m = new CPModel();
// data
int n = 11 + 3; //number of tasks (include the three fake tasks)
int[] heights_data = new int[]{2, 1, 4, 2, 3, 1, 5, 6, 2, 1, 3, 1, 1, 2};
int[] durations_data = new int[]{1, 1, 1, 2, 1, 3, 1, 1, 3, 4, 2, 3, 1, 1};
// variables
IntegerVariable capa = constant(7);
IntegerVariable[] starts = makeIntVarArray("start", n, 0, 5, "cp:bound");
IntegerVariable[] ends = makeIntVarArray("end", n, 0, 6, "cp:bound");
IntegerVariable[] duration = new IntegerVariable[n];
IntegerVariable[] height = new IntegerVariable[n];
for (int i = 0; i < height.length; i++) {
    duration[i] = constant(durations_data[i]);
    height[i] = makeIntVar("height " + i, new int[]{0, heights_data[i]});
}
IntegerVariable[] bool = makeIntVarArray("taskIn?", n, 0, 1);
IntegerVariable obj = makeIntVar("obj", 0, n, "cp:bound", "cp:objective");
//post the cumulative
m.addConstraint(cumulative(starts, ends, duration, height, capa, ""));
//post the channeling to know if the task is scheduled or not
for (int i = 0; i < n; i++) {
    m.addConstraint(boolChanneling(bool[i], height[i], heights_data[i]));
}
//state the objective function
m.addConstraint(eq(sum(bool), obj));
CPSolver s = new CPSolver();
s.read(m);
//set the fake tasks to establish the profile capacity of the ressource
try {
    s.getVar(starts[0]).setVal(1);
    s.getVar(ends[0]).setVal(2);
    s.getVar(height[0]).setVal(2);
    s.getVar(starts[1]).setVal(2);
    s.getVar(ends[1]).setVal(3);
    s.getVar(height[1]).setVal(1);
    s.getVar(starts[2]).setVal(3);
    s.getVar(ends[2]).setVal(4);
    s.getVar(height[2]).setVal(4);
} catch (ContradictionException e) {
    System.out.println("error, no contradiction expected at this stage");
}
// maximize the number of tasks placed in this profile
s.maximize(s.getVar(obj), false);
System.out.println("Objective : " + (s.getVar(obj).getVal() - 3));
for (int i = 3; i < starts.length; i++) {
    if (s.getVar(height[i]).getVal() != 0)
```

```
            System.out.println("[" + s.getVar(starts[i]).getVal() + "␣-␣"
                    + (s.getVar(ends[i]).getVal() - 1) + "]:"
                    + s.getVar(height[i]).getVal());
        }
```

9.7 disjunctive (constraint)

`disjunctive(start,duration)` states that a set of tasks (defined by their starting times and durations) are executed on a ddisjunctive resource, i.e. they do not overlap in time:

$$|\{i \mid \texttt{start}[i] \leq t < \texttt{start}[i] + \texttt{duration}[i]\}| \leq 1, \quad (\forall \text{ time } t)$$

The notion of task does not exist yet in Choco. The `disjunctive` takes therefore as input arrays of integer variables (of same size n) denoting the starting and duration of each task. When the array of finishing times is also specified, the constraint ensures that `start[i] + duration[i] = end[i]` for all task i. As usual, a task is executed in the interval `[start,end-1]`.

- **API** :
  - `disjunctive(IntegerVariable[] start, int[] duration, String...options)`
  - `disjunctive(IntegerVariable[] start, IntegerVariable[] duration, String... options)`
  - `disjunctive(IntegerVariable[] start, IntegerVariable[] end, IntegerVariable[] duration , String... options)`
  - `disjunctive(IntegerVariable[] start, IntegerVariable[] end, IntegerVariable[] duration , IntegerVariable uppBound, String... options)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: disjunctive

Example: `//TODO: complete`

9.8 distanceEQ (constraint)

`distanceEQ`(x_1, x_2, x_3, c) states that x_3 plus an offset c (by default $c = 0$) is equal to the distance between x_1 and x_2:

$$x_3 + c = |x_1 - x_2|$$

- **API** :
  - `distanceEQ(IntegerVariable x1, IntegerVariable x2, int x3)`
  - `distanceEQ(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3)`
  - `distanceEQ(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3, int c)`

- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: `all_min_dist` (variant)

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v0 = makeIntVar("v0", 0, 5);
IntegerVariable v1 = makeIntVar("v1", 0, 5);
IntegerVariable v2 = makeIntVar("v2", 0, 5);
m.addConstraint(distanceEQ(v0, v1, v2, 0));
s.read(m);
s.solveAll();
```

9.9 distanceGT (constraint)

`distanceGT`(x_1, x_2, x_3, c) states that x_3 plus an offset c (by default $c = 0$) is strictly greater than the distance between x_1 and x_2:

$$x_3 + c > |x_1 - x_2|$$

- **API** :
  - `distanceGT(IntegerVariable x1, IntegerVariable x2, int x3)`
  - `distanceGT(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3)`
  - `distanceGT(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3, int c)`
- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: `all_min_dist` (variant)

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v0 = makeIntVar("v0", 0, 5);
IntegerVariable v1 = makeIntVar("v1", 0, 5);
IntegerVariable v2 = makeIntVar("v2", 0, 5);
m.addConstraint(distanceGT(v0, v1, v2, 0));
s.read(m);
s.solveAll();
```

9.10 distanceLT (constraint)

distanceLT(x_1, x_2, x_3, c) states that x_3 plus an offset c (by default $c = 0$) is strictly smaller than the distance between x_1 and x_2:

$$x_3 + c < |x_1 - x_2|$$

- **API** :
  - `distanceLT(IntegerVariable x1, IntegerVariable x2, int x3)`
  - `distanceLT(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3)`
  - `distanceLT(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3, int c)`
- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v0 = makeIntVar("v0", 0, 5);
IntegerVariable v1 = makeIntVar("v1", 0, 5);
IntegerVariable v2 = makeIntVar("v2", 0, 5);
m.addConstraint(distanceLT(v0, v1, v2, 0));
s.read(m);
s.solveAll();
```

9.11 distanceNEQ (constraint)

distanceNEQ(x_1, x_2, x_3, c) states that x_3 plus an offset c (by default $c = 0$) is not equal to the distance between x_1 and x_2:

$$x_3 + c \neq |x_1 - x_2|$$

- **API** :
  - `distanceNEQ(IntegerVariable x1, IntegerVariable x2, int x3)`
  - `distanceNEQ(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3)`
  - `distanceNEQ(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3, int c)`
- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: all_min_dist (variant)

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v0 = makeIntVar("v0", 0, 5);
IntegerVariable v1 = makeIntVar("v1", 0, 5);
m.addConstraint(distanceNEQ(v0, v1, 0));
s.read(m);
s.solveAll();
```

9.12 eq (constraint)

eq(x, y) states that the two arguments are equal:

$$x = y$$

- **API** :
  - eq(IntegerExpressionVariable x, IntegerExpressionVariable y)
  - eq(IntegerExpressionVariable x, int y)
  - eq(int x, IntegerExpressionVariable y)
  - eq(SetVariable x, SetVariable y)
  - eq(RealExpressionVariable x, RealExpressionVariable y)
  - eq(RealExpressionVariable x, double y)
  - eq(double x, RealExpressionVariable y)
  - eq(IntegerVariable x, RealVariable y)
  - eq(RealVariable x, IntegerVariable y)
- **return type** : Constraint
- **options** : *n/a*
- **favorite domain** : *to complete.*
- **references** :
  global constraint catalog: eq (on domain variables) and eq_set (on set variables).

Examples:

- example1:

```
Model m = new CPModel();
Solver s = new CPSolver();
int c = 1;
IntegerVariable v = makeIntVar("v", 0, 2);
m.addConstraint(eq(v, c));
s.read(m);
s.solve();
```

- example2

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 2);
IntegerVariable v2 = makeIntVar("v2", 0, 2);
IntegerExpressionVariable w1 = plus(v1, 1);
IntegerExpressionVariable w2 = minus(v2, 1);
m.addConstraint(eq(w1, w2));
s.read(m);
s.solve();
```

9.13 eqCard (constraint)

`eqCard`(s, x) states that the cardinality of set s is equal to x:

$$|s| = x$$

- **API** :
  - `eqCard(SetVariable s, IntegerVariable x)`
  - `eqCard(SetVariable s, int x)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable set = makeSetVar("s", 1, 5);
IntegerVariable card = makeIntVar("card", 2, 3);
m.addConstraint(member(set, 3));
m.addConstraint(eqCard(set, card));
s.read(m);
s.solve();
```

9.14 equation (constraint)

`equation`(x, c, z) states a linear equation:

$$c_1x_1 + c_2x_2 + ... + c_nx_n = z$$

It enforces GAC using regular to state a *knapsack* constraint.

- **API** :
  - equation(IntegerVariable[] x, int[] c, int z)
- **return type** : `Constraint`

- **options** : *n/a*
- **favorite domain** : *to complete*

Example:

```
CPModel m = new CPModel();
CPSolver s = new CPSolver();
int n = 10;
IntegerVariable[] bvars = makeIntVarArray("b", n, 0, 10, "cp:enum");
int[] coefs = new int[n];

int charge = 10;
Random rand = new Random();
for (int i = 0; i < coefs.length; i++) {
    coefs[i] = rand.nextInt(10);
}
Constraint knapsack = equation(bvars, coefs, charge);
m.addConstraint(knapsack);
s.read(m);
s.solveAll();
```

9.15 FALSE (constraint)

FALSE always returns *false*.

9.16 feasPairAC (constraint)

`feasPairAC`$(x, y, feasTuples)$ states an extensional binary constraint on (x, y) defined by the table $feasTuples$ of compatible pairs of values, and then enforces arc consistency. Two APIs are available to define the compatible pairs:

- if $feasTuples$ is encoded as a list of pairs `List<int[2]>`, then:

$$\exists \text{ tuple } i \mid \quad (x, y) = feasTuples[i]$$

- if $feasTuples$ is encoded as a boolean matrix `boolean[][]`, let $\underline{x}$ and $\underline{y}$ be the initial minimum values of x and y, then:

$$\exists (u, v) \mid \quad (x, y) = (u + \underline{x}, v + \underline{y}) \ \wedge \ feasTuples[u][v]$$

The two APIs are duplicated to allow definition of options.

- **API** :
  - `feasPairAC(IntegerVariable x, IntegerVariable y, List<int[]> feasTuples)`
  - `feasPairAC(String options, IntegerVariable x, IntegerVariable y, List<int[]> feasTuples)`
  - `feasPairAC(IntegerVariable x, IntegerVariable y, boolean[][] feasTuples)`
  - `feasPairAC(String options, IntegerVariable x, IntegerVariable y, boolean[][] feasTuples)`
- **return type** : `Constraint`
- **options** :

- *no option*: use AC3 (default arc consistency)
- `cp:ac3`: to get AC3 algorithm (searching from scratch for supports on all values)
- `cp:ac2001`: to get AC2001 algorithm (maintaining the current support of each value)
- `cp:ac32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
- `cp:ac322`: to get AC3 with the used of `BitSet` to know if a support still exists

- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: in_relation

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
ArrayList couples2 = new ArrayList();
couples2.add(new int[]{1, 2});
couples2.add(new int[]{1, 3});
couples2.add(new int[]{2, 1});
couples2.add(new int[]{3, 1});
couples2.add(new int[]{4, 1});
IntegerVariable v1 = makeIntVar("v1", 1, 4);
IntegerVariable v2 = makeIntVar("v2", 1, 4);
m.addConstraint(feasPairAC("cp:ac32", v1, v2, couples2));
s.read(m);
s.solveAll();
```

9.17 feasTupleAC (constraint)

`feasTupleAC`$(x, feasTuples)$ states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the table $feasTuples$ of compatible tuples of values, and then enforces arc consistency:

$$\exists \text{ tuple } i \mid \quad (x_1, \ldots, x_n) = feasTuples[i]$$

The API is duplicated to define options.

- **API** :
  - `feasTupleAC(List<int[]> feasTuples, IntegerVariable... x)`
  - `feasTupleAC(String options, List<int[]> feasTuples, IntegerVariable... x)`
- **return type**: `Constraint`
- **options** :
  - *no option*: use AC32 (default arc consistency)
  - `cp:ac32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `cp:ac2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `cp:ac2008`: to get AC2008 algorithm (maintained by STR)
- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: in_relation

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 2);
IntegerVariable v2 = makeIntVar("v2", 0, 4);
ArrayList feasTuple = new ArrayList();
feasTuple.add(new int[]{1, 1}); // x*y = 1
feasTuple.add(new int[]{2, 4}); // x*y = 1
m.addConstraint(feasTupleAC("cp:ac2001", feasTuple, new IntegerVariable[]{v1, v2}));
s.read(m);
s.solve();
```

9.18 feasTupleFC (constraint)

`feasTupleFC`$(x, feasTuples)$ states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the table $feasTuples$ of compatible tuples of values, and then performs Forward Checking:

$$\exists \text{ tuple } i \mid \quad (x_1, \ldots, x_n) = feasTuples[i]$$

- **API** : `feasTupleFC(List<int[]> tuples, IntegerVariable... x)`
- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain**: *to complete*
- **references** :
  global constraint catalog: in_relation

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 2);
IntegerVariable v2 = makeIntVar("v2", 0, 4);
ArrayList feasTuple = new ArrayList();
feasTuple.add(new int[]{1, 1}); // x*y = 1
feasTuple.add(new int[]{2, 4}); // x*y = 1
m.addConstraint(feasTupleFC(feasTuple, new IntegerVariable[]{v1, v2}));
s.read(m);
s.solve();
```

9.19 geost (constraint)

`geost` is a global constraint that generically handles a variety of geometrical placement problems. It handles geometrical constraints (non-overlapping, distance, etc.) between polymorphic objects (ex: polymorphism can be used for representing rotation) in any dimension. The parameters of `geost`($dim, objects, shiftedBoxes, eCtrs$) are respectively: the space dimension, the list of geometrical objects, the set of boxes that compose the shapes of the objects, the set of geometrical constraints.

- **API** :
```
geost(int dim, Vector<GeostObject> objects, Vector<ShiftedBox> shiftedBoxes, Vector<ExternalConstraint
> eCtrs)
geost(int dim, Vector<GeostObject> objects, Vector<ShiftedBox> shiftedBoxes, Vector<ExternalConstraint
> eCtrs, Vector<int[]> ctrlVs)
```

- **return type** : `Constraint`

- **options** :*n/a*

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: geost

The geost constraint requires the creation of different objects:

parameter	type	description
objects	`Vector<GeostObject>`	geometrical objects
shiftedBoxes	`Vector<ShiftedBox>`	boxes that compose the object shapes
eCtrs	`Vector<ExternalConstraint>`	geometrical constraints
ctrlVs	`Vector<int[]>`	controlling vectors (for greedy mode)

Where a **GeostObject** is defined by:

attribute	type	description
dim	`int`	dimension
objectId	`int`	object id
shapeId	`IntegerVariable`	shape id
coordinates	`IntegerVariable[`*dim*`]`	coordinates of the origin
startTime	`IntegerVariable`	starting time
durationTime	`IntegerVariable`	duration
endTime	`IntegerVariable`	finishing time

Where a **ShiftedBox** is a *dim*-box defined by the shape it belongs to, its origin (the coordinates of the lower left corner of the box) and its lengths in every dimensions:

attribute	type	description
sid	`int`	shape id
offset	`int[`*dim*`]`	coordinates of the offset (lower left corner)
size	`int[`*dim*`]`	lengths in every dimensions

Where an **ExternalConstraint** contains informations and functionality common to all external constraints and is defined by:

attribute	type	description
ectrID	`int`	constraint id
dimensions	`int[]`	list of dimensions that the external constraint is active for
objectIdentifiers	`int[]`	list of object ids that this external constraint affects.

For further informations, visit the following page.

Example:

```
Model m = new CPModel();
int dim = 3;
int lengths[] = {5, 3, 2};
int widths[] = {2, 2, 1};
int heights[] = {1, 1, 1};
int nbOfObj = 3;
long seed = 0;
//Create the Objects
Vector<GeostObject> obj = new Vector<GeostObject>();
for (int i = 0; i < nbOfObj; i++) {
    IntegerVariable shapeId = Choco.makeIntVar("sid", i, i);
    IntegerVariable coords[] = new IntegerVariable[dim];
    for (int j = 0; j < coords.length; j++) {
        coords[j] = Choco.makeIntVar("x" + j, 0, 2);
    }
    IntegerVariable start = Choco.makeIntVar("start", 1, 1);
    IntegerVariable duration = Choco.makeIntVar("duration", 1, 1);
    IntegerVariable end = Choco.makeIntVar("end", 1, 1);
    obj.add(new GeostObject(dim, i, shapeId, coords, start, duration, end));
}
//Create the ShiftedBoxes and add them to corresponding shapes
Vector<ShiftedBox> sb = new Vector<ShiftedBox>();
int[] t = {0, 0, 0};
for (int d = 0; d < nbOfObj; d++) {
    int[] l = {lengths[d], heights[d], widths[d]};
    sb.add(new ShiftedBox(d, t, l));
}
//Create the external constraints vector
Vector<IExternalConstraint> ectr = new Vector<IExternalConstraint>();
//create the list of dimensions for the external constraint
int[] ectrDim = new int[dim];
for (int d = 0; d < dim; d++)
    ectrDim[d] = d;
//create the list of object ids for the external constraint
int[] objOfEctr = new int[nbOfObj];
for (int d = 0; d < nbOfObj; d++) {
    objOfEctr[d] = obj.elementAt(d).getObjectId();
}
//create and add one external constraint of type non overlapping
NonOverlappingModel n = new NonOverlappingModel(Constants.NON_OVERLAPPING, ectrDim,
     objOfEctr);
ectr.add(n);
//create and post the geost constraint
Constraint geost = Choco.geost(dim, obj, sb, ectr);
m.addConstraint(geost);
Solver s = new CPSolver();
```

```
s.read(m);
s.setValIntSelector(new RandomIntValSelector(seed));
s.setVarIntSelector(new RandomIntVarSelector(s, seed));
s.solveAll();
```

9.20 geq (constraint)

geq(x, y) states that x is greater than or equal to y:

$$x \geq y$$

- **API** :
  - geq(IntegerExpressionVariable x, IntegerExpressionVariable y)
  - geq(IntegerExpressionVariable x, int y)
  - geq(int x, IntegerExpressionVariable y)
  - geq(RealExpressionVariable x, RealExpressionVariable y)
  - geq(RealExpressionVariable x, double y)
  - geq(double x, RealExpressionVariable y)
- **return type** : Constraint
- **options** : *n/a*
- **favorite domain** : *to complete.*
- **references** :
  global constraint catalog: geq

Examples:

- example1:

```
Model m = new CPModel();
Solver s = new CPSolver();
int c = 1;
IntegerVariable v = makeIntVar("v", 0, 2);
m.addConstraint(eq(v, c));
s.read(m);
s.solve();
```

- example2

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 2);
IntegerVariable v2 = makeIntVar("v2", 0, 2);
IntegerExpressionVariable w1 = plus(v1, 1);
IntegerExpressionVariable w2 = minus(v2, 1);
m.addConstraint(eq(w1, w2));
s.read(m);
s.solve();
```

9.21 geqCard (constraint)

`geqCard`(s, x) states that the cardinality of set s is greater than or equal to x:

$$|s| \geq x$$

- **API** :
  - `geqCard(SetVariable s, IntegerVariable x)`
  - `geqCard(SetVariable s, int x)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable set = makeSetVar("s", 1, 5);
IntegerVariable i = makeIntVar("card", 2, 3);
m.addConstraint(member(set, 3));
m.addConstraint(geqCard(set, i));
s.read(m);
s.solve();
```

9.22 globalCardinality (constraint)

`globalCardinality`(x, low, up) states bounds on the occurrence numbers of any value v in x (here, offset min is the minimum value over all variables in x) :

$$low[v - min] \leq |\{i \mid x_i = v\}| \leq up[v - min], \quad \forall \text{ value } v$$

Mulitple APIs exist:

- *bounds on cardinalities:* Given an array of variables x, min the minimal value over all variables, and max the maximal value over all variables, the constraint ensures that the number of occurrences of value v among the variables is between $low[v - min]$ and $up[v - min]$. Note that the length of low and up should be $max - min + 1$. Use the propagator of [**?**].
- *default offset* $min = 1$*:* Given an array of variables x, the constraint ensures that the number of occurrences of the value 1 in all the variables x is between $low[0]$ and $up[0]$, and generally the number of occurrences of the value v in x is between $low[v - 1]$ and $up[v - 1]$.
- *variable cardinalities:* Given an array of variables x, an array of variables *card* to represent the cardinalities, the constraint ensures that the number of occurrences of the value v among the variables is equal to $card[v]$. This constraint:
  - enforces Bound Consistency over x regarding the lower and upper bounds of *card*,

- maintains the upper bounds of *card* by counting the number of variables in which each value can occur,
- maintains the lower bounds of *card* by counting the number of variables instantiated to each value,
- enforces $card[0] + \cdots + card[m] = n$, where n is the number of variables and m the number of values.

The APIs are duplicated to define options.

- **API** :
  - `globalCardinality(IntegerVariable[] x, int min, int max, int[] low, int[] up)`
  - `globalCardinality(String options, IntegerVariable[] x, int min, int max, int[] low, int [] up)`
  - `globalCardinality(IntegerVariable[] x, int[] low, int[] up)`
  - `globalCardinality(String options, IntegerVariable[] x, int[] low, int[] up)`
  - `globalCardinality(IntegerVariable[] x, int min, int max, IntegerVariable[] card)`
- **return type** : `Constraint`
- **options**:
  - *no option* :
  - `cp:ac` : for [?] implementation of arc consistency
  - `cp:bc` : for [?] implementation of bound consistency
- **favorite domain** : *to complete*
- **references** :
  - [?]: *Generalized arc consistency for global cardinality constraint*,
  - [?]: *An efficient bounds consistency algorithm for the global cardinality constraint*
  - global constraint catalog: global_cardinality

Example:

```
int n = 5;
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable[] vars = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
   vars[i] = makeIntVar("var␣" + i, 1, n);
}
int[] LB2 = {0, 1, 1, 0, 3};
int[] UB2 = {0, 1, 1, 0, 3};
m.addConstraint("cp:bc", globalCardinality(vars, 1, n, LB2, UB2));
s.read(m);
s.solve();
```

9.23 gt (constraint)

`gt`(x, y) states that x is strictly greater than y:

$$x > y$$

- **API** :
  - `gt(IntegerExpressionVariable x, IntegerExpressionVariable y)`
  - `gt(IntegerExpressionVariable x, int y)`
  - `gt(int x, IntegerExpressionVariable y)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete.*
- **references** :
  global constraint catalog: gt

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int c = 1;
IntegerVariable v = makeIntVar("v", 0, 2);
m.addConstraint(gt(v, c));
s.read(m);
s.solve();
```

9.24 ifOnlyIf (constraint)

`ifOnlyIf`(c_1, c_2) states that c_1 holds if and only if c_2 holds:

$$c_1 \iff c_2$$

- **API** : `ifOnlyIf(Constraint c1, Constraint c2)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 3);
IntegerVariable y = makeIntVar("y", 1, 3);
IntegerVariable z = makeIntVar("z", 1, 3);
m.addVariables("cp:bound",x ,y, z);
m.addConstraint(ifOnlyIf(lt(x, y), lt(y, z)));
s.read(m);
s.solveAll();
```

9.25 ifThenElse (constraint)

`ifThenElse`(c_1, c_2, c_3) states that if c_1 holds then c_2 holds, otherwise c_3 holds:

$$(c_1 \wedge c_2) \vee (\neg c_1 \wedge c_3)$$

`ifThenElse`(c_1, c_2, c_3) can also state that if the first constraint is satisfied, it returns the second parameter, otherwise it returns the the third one.

- **API** :
  - `ifThenElse(Constraint c1, Constraint c2, Constraint c3)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 3);
IntegerVariable y = makeIntVar("y", 1, 3);
IntegerVariable z = makeIntVar("z", 1, 3);
// use API ifThenElse(Constraint, Constraint, Constraint)
m.addConstraint(ifThenElse(lt((x), (y)), gt((y), (z)), FALSE));
 // and ifThenElse(Constraint, IntegerExpressionVariable, IntegerExpressionVariable)
m.addConstraint(leq(z, ifThenElse(lt(x, y), constant(1), plus(x,y))));
s.read(m);
s.solveAll();
```

9.26 implies (constraint)

`implies`(c_1, c_2) states that if c_1 holds then c_2 holds:

$$c_1 \implies c_2$$

- **API** : `implies(Constraint c1, Constraint c2)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 2);
IntegerVariable y = makeIntVar("y", 1, 2);
IntegerVariable z = makeIntVar("z", 1, 2);
```

```
m.addVariables("cp:bound",x ,y, z);
Constraint e1 = implies(leq(x, y), leq(x, z));
m.addConstraint(e1);
s.read(m);
s.solveAll();
```

9.27 infeasPairAC (constraint)

`infeasPairAC`$(x, y, infeasTuples)$ states an extensional binary constraint on (x, y) defined by the table $infeasTuples$ of forbidden pairs of values, and then enforces arc consistency. Two APIs are available to define the forbidden pairs:

- if $infeasTuples$ is encoded as a list of pairs `List<int[2]>`, then:

$$\forall \text{ tuple } i \mid \quad (x, y) \neq infeasTuples[i]$$

- if $infeasTuples$ is encoded as a boolean matrix `boolean[][]`, let $\underline{x}$ and $\underline{y}$ be the initial minimum values of x and y, then:

$$\forall (u, v) \mid \quad (x, y) = (u + \underline{x}, v + \underline{y}) \ \vee \ \neg infeasTuples[u][v]$$

The two APIs are duplicated to allow definition of options.

- **API** :
  - `infeasPairAC(IntegerVariable x, IntegerVariable y, List<int[]> infeasTuples)`
  - `infeasPairAC(String options, IntegerVariable x, IntegerVariable y, List<int[]> infeasTuples )`
  - `infeasPairAC(IntegerVariable x, IntegerVariable y, boolean[][] infeasTuples)`
  - `infeasPairAC(String options, IntegerVariable x, IntegerVariable y, boolean[][] infeasTuples )`
- **return type** : `Constraint`
- **options** :
  - *no option*: use AC3 (default arc consistency)
  - `cp:ac3`: to get AC3 algorithm (searching from scratch for supports on all values)
  - `cp:ac2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `cp:ac32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `cp:ac322`: to get AC3 with the used of `BitSet` to know if a support still exists
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
boolean[][] matrice2 = new boolean[][]{
            {false, true, true, false},
            {true, false, false, false},
            {false, false, true, false},
            {false, true, false, false}};
```

```
IntegerVariable v1 = makeIntVar("v1", 1, 4);
IntegerVariable v2 = makeIntVar("v2", 1, 4);
m.addConstraint(feasPairAC("cp:ac32",v1, v2, matrice2));
s.read(m);
```

9.28 infeasTupleAC (constraint)

`infeasTupleAC`($x, feasTuples$) states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the table $infeasTuples$ of compatible tuples of values, and then enforces arc consistency:

$$\forall \text{ tuple } i \mid \quad (x_1, \ldots, x_n) \neq infeasTuples[i]$$

The API is duplicated to define options.

- **API** :
  - `infeasTupleAC(List<int[]> infeasTuples, IntegerVariable... x)`
  - `infeasTupleAC(String options, List<int[]> infeasTuples, IntegerVariable... x)`
- **return type**: `Constraint`
- **options** :
  - *no option*: use AC32 (default arc consistency)
  - `cp:ac32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `cp:ac2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `cp:ac2008`: to get AC2008 algorithm (maintained by STR)
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 1, 5);
IntegerVariable z = makeIntVar("z", 1, 5);
ArrayList forbiddenTuples = new ArrayList();
forbiddenTuples.add(new int[]{1, 1, 1});
forbiddenTuples.add(new int[]{2, 2, 2});
forbiddenTuples.add(new int[]{2, 5, 3});
m.addConstraint(infeasTupleAC(forbiddenTuples, new IntegerVariable[]{x, y, z}));
s.read(m);
s.solveAll();
```

9.29 infeasTupleFC (constraint)

`infeasTupleFC`($x, feasTuples$) states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the table $infeasTuples$ of compatible tuples of values, and then performs Forward Checking:

$$\forall \text{ tuple } i \mid \quad (x_1, \ldots, x_n) \neq infeasTuples[i]$$

- **API** : `infeasTupleFC(List<int[]> infeasTuples, IntegerVariable... x)`
- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain**: *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 1, 5);
IntegerVariable z = makeIntVar("z", 1, 5);
ArrayList forbiddenTuples = new ArrayList();
forbiddenTuples.add(new int[]{1, 1, 1});
forbiddenTuples.add(new int[]{2, 2, 2});
forbiddenTuples.add(new int[]{2, 5, 3});
m.addConstraint(infeasTupleFC(forbiddenTuples, new IntegerVariable[]{x, y, z}));
s.read(m);
s.solveAll();
```

9.30 intDiv (constraint)

`intDiv`(x, y, z) states that the z is equal to the integer quotient of x by y:

$$z = \lfloor x/y \rfloor$$

- **API**: `intDiv(IntegerVariable x, IntegerVariable y, IntegerVariable z)`
- **return type** : `Constraint`
- **option** : *n/a*
- **favorite domain**: bound

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
long seed = 0;
IntegerVariable x = makeIntVar("x", 3, 5);
IntegerVariable y = makeIntVar("y", 1, 2);
IntegerVariable z = makeIntVar("z", 0, 5);
m.addConstraint(intDiv(x, y, z));
s.setVarIntSelector(new RandomIntVarSelector(s, seed));
s.setValIntSelector(new RandomIntValSelector(seed + 1));
s.read(m);
s.solve();
```

9.31 inverseChanneling (constraint)

`inverseChanneling`(x, y) states a channeling between two arrays x and y of integer variables with the same domain.It enforces that if the i-th element of x is equal to j then the j-th element of y is equal to i and conversely:

$$x_i = j \iff y_j = i$$

- **API** : `inverseChanneling(IntegerVariable[] x, IntegerVariable[] y)`
- **return type** : `Constraint`
- **options** : *no options*
- **favorite domain** : enumerated for x
- **references** :
  global constraint catalog: inverse

Example:

```
int n = 8;
Model m = new CPModel();
IntegerVariable[] queens = new IntegerVariable[n];
IntegerVariable[] queensdual = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queens[i] = makeIntVar("Q" + i, 1, n);
    queensdual[i] = makeIntVar("QD" + i, 1, n);
}
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
       int k = j - i;
       m.addConstraint(neq(queens[i], queens[j]));
       m.addConstraint(neq(queens[i], plus(queens[j], k))); // diagonal constraints
       m.addConstraint(neq(queens[i], minus(queens[j], k))); // diagonal constraints
    }
}
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        int k = j - i;
        m.addConstraint(neq(queensdual[i], queensdual[j]));
        m.addConstraint(neq(queensdual[i], plus(queensdual[j], k))); // diagonal
            constraints
        m.addConstraint(neq(queensdual[i], minus(queensdual[j], k))); // diagonal
            constraints
    }
}
m.addConstraint(inverseChanneling(queens, queensdual));
m.addVariables("cp:decision", queens);
Solver s = new CPSolver();
s.read(m);
s.solveAll();
```

9.32 isIncluded (constraint)

`isIncluded`(x, y) states that the second set y contains the first set x:

$$x \subseteq y$$

- **API** : `isIncluded(SetVariable x, SetVariable y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable v1 = makeSetVar("v1", 3, 4);
SetVariable v2 = makeSetVar("v2", 3, 8);
m.addConstraint(isIncluded(v1, v2));
s.read(m);
s.solveAll();
```

9.33 isNotIncluded (constraint)

`isNotIncluded`(x, y) states that the second set y does not contain the first set x:

$$x \not\subseteq y$$

- **API** : `isNotIncluded(SetVariable x, SetVariable y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable v1 = makeSetVar("v1", 3, 4);
SetVariable v2 = makeSetVar("v2", 3, 8);
m.addConstraint(isNotIncluded(v1, v2));
s.read(m);
s.solveAll();
```

9.34 leq (constraint)

`leq`(x, y) states that x is less than or equal to y:

$$x \leq y$$

- **API** :
  - `leq(IntegerExpressionVariable x, IntegerExpressionVariable y)`
  - `leq(IntegerExpressionVariable x, int y)`
  - `leq(int x, IntegerExpressionVariable y)`
  - `leq(RealExpressionVariable x, RealExpressionVariable y)`
  - `leq(RealExpressionVariable x, double y)`
  - `leq(double x, RealExpressionVariable y)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete.*
- **references** :
  global constraint catalog: leq

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int c = 1;
IntegerVariable v = makeIntVar("v", 0, 2);
m.addConstraint(leq(v, c));
IntegerVariable v1 = makeIntVar("v1", 0, 2);
IntegerVariable v2 = makeIntVar("v2", 0, 2);
IntegerExpressionVariable w1 = plus(v1, 1);
IntegerExpressionVariable w2 = minus(v2, 1);
m.addConstraint(leq(w1, w2));
s.read(m);
s.solve();
```

9.35 leqCard (constraint)

`leqCard`(s, x) states that the cardinality of set s is less than or equal to x:

$$|s| \leq x$$

- **API** :
  - `leqCard(SetVariable s, IntegerVariable x)`
  - `leqCard(SetVariable s, int x)`

- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable set = makeSetVar("s", 1, 5);
IntegerVariable i = makeIntVar("card", 2, 3);
m.addConstraint(member(set, 3));
m.addConstraint(leqCard(set, i));
s.read(m);
s.solve();
```

9.36 lex (constraint)

`lex`(x, y) enforces a strict lexicographic ordering $x <_{lex} y$ between two arrays of same size n:

$$\exists\, j \in \{1, \ldots, n\} \mid \quad x_j < y_j \quad \wedge \quad x_i = y_i \ (\forall\, i < j)$$

- **API** : `lex(IntegerVariable[] x, IntegerVariable[] y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  - [**?**]: *Global Constraints for Lexicographic Orderings*
  - global constraint catalog: lex_less

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int n1 = 8;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n1 / 2];
IntegerVariable[] vs2 = new IntegerVariable[n1 / 2];
for (int i = 0; i < n1 / 2; i++) {
   vs1[i] = makeIntVar("" + i, 0, k);
   vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lex(vs1, vs2));
s.read(m);
s.solve();
```

9.37 lexChain (constraint)

`lexChain`$(x^1, x^2, x^3, \ldots)$ enforces a strict lexicographic ordering on a chain of integer vectors:

$$x^1 <_{lex} x^2 <_{lex} x^3 <_{lex} \cdots$$

- **API** : `lexChain(IntegerVariable[]... arrayOfVectors)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*
- **references** :
  - [?] *Arc-Consistency for a chain of Lexicographic Ordering Constraints*
  - global constraint catalog: lex_chain_less

9.38 lexChainEq (constraint)

`lexChainEq`$(x^1, x^2, x^3, \ldots)$ enforces a lexicographic ordering on a chain of integer vectors:

$$x^1 \leq_{lex} x^2 \leq_{lex} x^3 \leq_{lex} \cdots$$

- **API** : `lexChainEq(IntegerVariable[]... arrayOfVectors)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*
- **references** :
  - [?] *Arc-Consistency for a chain of Lexicographic Ordering Constraints*
  - global constraint catalog: lex_chain_lesseq

Example:

```
CPModel m = new CPModel();
CPSolver s = new CPSolver();
```

9.39 lexeq (constraint)

`lexeq`(x, y) enforces a lexicographic ordering $x \leq_{lex} y$ between two arrays of same size n:

$$\exists\, j \in \{1, \ldots, n\} \mid \quad x_j \leq y_j \quad \wedge \quad x_i = y_i \ (\forall\, i < j)$$

- **API** : `lexeq(IntegerVariable[] x, IntegerVariable[] y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  - [**?**]: *Global Constraints for Lexicographic Orderings*
  - global constraint catalog: lex_lesseq

Example:

```
Model pb = new CPModel();
Solver s = new CPSolver();

int n1 = 8;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n1 / 2];
IntegerVariable[] vs2 = new IntegerVariable[n1 / 2];
for (int i = 0; i < n1 / 2; i++) {
   vs1[i] = makeIntVar("" + i, 0, k);
   vs2[i] = makeIntVar("" + i, 0, k);
}

m.addConstraint(lexeq(vs1, vs2));
s.read(m);
s.solve();
```

9.40 leximin (constraint)

TODO: verify the specifications of the implemented version.

Let $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ be two vectors of n integers, and let x' and y' be respectively permutations of vectors x and y sorted by increasing order of the components. Constraint `leximin(x, y)` holds if and only if $x' <_{lex} y'$:

$$\exists\, j \in \{1, \ldots, n\} \mid \quad x'_j < y'_j \quad \wedge \quad x'_i = y'_i \ (\forall\, i < j)$$

- **API** :
  - `leximin(IntegerVariable[] x, IntegerVariable[] y)`
  - `leximin(int[] x, IntegerVariable[] y)`

- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  - [?]: *Multiset ordering constraints*
  - global constraint catalog: lex_lesseq_allperm (variant)

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();

IntegerVariable[] u = makeIntVarArray("u", 3, 2, 5);
IntegerVariable[] v = makeIntVarArray("v", 3, 2, 4);
m.addConstraint(leximin(u, v));
m.addConstraint(allDifferent(v));

s.read(m);
s.solve();
```

9.41 lt (constraint)

`lt`(x, y) states that x is strictly smaller than y:

$$x < y$$

- **API** :
  - `lt(IntegerExpressionVariable x, IntegerExpressionVariable y)`
  - `lt(IntegerExpressionVariable x, int y)`
  - `lt(int x, IntegerExpressionVariable y)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete.*
- **references** :
  global constraint catalog: lt

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int c = 1;
IntegerVariable v = makeIntVar("v", 0, 2);
m.addConstraint(lt(v, c));
s.read(m);
s.solve();
```

9.42 max (constraint)

9.42.1 max of a list

`max`(x, z) states that z is equal to the greater element of vector x:

$$z = \max(x_1, x_2, ..., x_n)$$

- **API**:
  - `max(IntegerVariable[] x, IntegerVariable z)`
  - `max(IntegerVariable x1, IntegerVariable x2, IntegerVariable z)`
  - `max(int x1, IntegerVariable x2, IntegerVariable z)`
  - `max(IntegerVariable x1, int x2, IntegerVariable z)`
- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: maximum

Example:

```
Model m = new CPModel();
Solver s= new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 1, 5);
IntegerVariable z = makeIntVar("z", 1, 5);
m.addVariable("cp:bound", x, y, z);
m.addConstraint(max(y, z, x));
s.read(m);
s.solve();
```

9.42.2 max of a set

`max`(s, x, z) states that z is equal to the greater element of vector x whose index is in set s:

$$z = \max_{i \in s}(x_i)$$

- **API**:
  - `max(SetVariable s, IntegerVariable[] x, IntegerVariable z)`
- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*

```
Model m = new CPModel();
Solver s= new CPSolver();
IntegerVariable[] x = constantArray(new int[]{5,7,9,10,12,3,2});
IntegerVariable max = makeIntVar("max", 1, 100);
SetVariable set = makeSetVar("set", 0, x.length-1);
m.addConstraints(max(set, x, max), leqCard(set, constant(5)));
s.read(m);
s.solve();
```

9.43 member (constraint)

`member`(x, s) states that integer x is contained in set s:

$$x \in s.$$

- **API** :
  - `member(int x, SetVariable s)`
  - `member(SetVariable s, int x)`
  - `member(SetVariable s, IntegerVariable x)`
  - `member(IntegerVariable x, SetVariable s)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: in_set

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int x = 3;
int card = 2;
SetVariable y = makeSetVar("y", 2, 4);
m.addConstraint(member(y, x));
m.addConstraint(eqCard(y, card));
s.read(m);
s.solveAll();
```

9.44 min (constraint)

9.44.1 min of a list

`mix`(x, z) states that z is equal to the smaller element of vector x:

$$z = \min(x_1, x_2, ..., x_n).$$

- **API**:
  - `min(IntegerVariable[] x, IntegerVariable z)`
  - `min(IntegerVariable x1, IntegerVariable x2, IntegerVariable z)`
  - `min(int x1, IntegerVariable x2, IntegerVariable z)`
  - `min(IntegerVariable x1, int x2, IntegerVariable z)`
- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: minimum

Example:

```
Model m = new CPModel();
Solver s= new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 1, 5);
IntegerVariable z = makeIntVar("z", 1, 5);
m.addVariable("cp:bound", x, y, z);
m.addConstraint(min(y, z, x));
s.read(m);
s.solve();
```

9.44.2 min of a set

`min`(s, x, z) states that z is equal to the smaller element of vector x whose index is in set s:

$$z = \min_{i \in s}(x_i).$$

- **API**:
  - `min(SetVariable s,IntegerVariable[] x, IntegerVariable z)`
- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*

```
Model m = new CPModel();
Solver s= new CPSolver();
IntegerVariable[] x = constantArray(new int[]{5,7,9,10,12,3,2});
IntegerVariable min = makeIntVar("min", 1, 100);
SetVariable set = makeSetVar("set", 0, x.length-1);
m.addConstraints(min(set, x, max), leqCard(set, constant(5)));
s.read(m);
s.solve();
```

9.45 mod (constraint)

mod(x_1, x_2, x_3) states that x_1 is congruent to x_2 modulo x_3:

$$x_1 \equiv x_2 \mod x_3$$

- **API** : `mod(IntegerVariable x1, IntegerVariable x2, int x3)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();

IntegerVariable x = makeIntVar("x", 0, 10);
IntegerVariable w = makeIntVar("w", 0, 10);

m.addConstraint(mod(w,x, 1));

s.read(m);
s.solve();
```

9.46 multiCostRegular (constraint)

multiCostRegular$(x, z, \mathcal{L}(\Pi), c)$ states that sequence x is a word belonging to the regular language $\mathcal{L}(\Pi)$,

$$(x_1, \ldots, x_n) \in \mathcal{L}(\Pi)$$

and that the bounded vector z is equal to the costs of x according to the assigment cost matrix c:

$$\sum_{i=1}^{n} c[r][i][x_i] = z[r], \quad \forall r \in \{0, \ldots, R\}$$

`multiCostRegular` is a conjunction of a `regular` constraint with $R+1$ cost functions. It may be used in the context of personnel scheduling problems, handling complex work regulations by the mean of regular expressions, together with cardinality or financial constraints by the mean of cost functions. The filtering algorithm associated with `multiCostRegular` is based on lagrangian relaxation and computations of shortest/longest pathes in a layered digraph [**?**]. It typically performs more filtering than the conjunction of `costRegular` and `globalCardinality` or than multiple `costRegular`.

The accepting language is specified by a deterministic finite automaton (DFA): Automaton Π is defined on a given *alphabet* $\Sigma \subseteq \mathbb{Z}$ by a set $Q = \{0, \ldots, m\}$ of *states*, a subset $A \subseteq Q$ of *final* or *accepting states* and a table $\Delta \subseteq Q \times \Sigma \times Q$ of *transitions* between states. Π is encoded as an object of class `Automaton`, whose API contains:

```
Automaton();
int addState();
void setStartingState(int state);
void setAcceptingState(int state);
void addTransition(int state1, int state2, int label);
int getNbStates();
```

The cost functions are encoded as one matrix `int cost[nTime][nAct][auto.getNbStates()][nRes]` such that `cost[i][j][s][r]` is the cost of assigning variable x_i to activity j at state s on dimension $r+1$.

- **API** :
  - `multiCostRegular(IntegerVariable[] x, IntegerVariable[] z, Automaton P, int[][][][] c)`
- **return type** : `Constraint`
- **options** :
  - `MultiCostRegular.DATA_STRUCT` is `MultiCostRegular.BITSET` or `MultiCostRegular.LIST`: a parameter stating which backtrable data structure to use for storing the outgoing arcs of the layered digraph. The observed behaviour is until 1000 arcs the bipartite list is much more efficient, afterwards the memory efficiency of the bitset representation allow faster operations.
  - `MultiCostRegular.U0`, `MultiCostRegular.R0`, `MultiCostRegular.MAXNONIMPROVEITER`, and `MultiCostRegular.MAXBOUNDITER` are value parameters of the subgradient algorithm used for solving the lagrangean relaxation.
  - `MultiCostRegular.D_PREC` is a double parameter stating the precision of float computation. It is set by default to 10^{-5}.
- **favorite domain** : *to complete*
- **references** :
  [**?**]: *Sequencing and Counting with the* `multicost-regular` *Constraint*

Example:

```
//1- create the model
Model m = new CPModel();

int nTime = 14; // 2 weeks: 14 days
int nAct = 3; // 3 activities: DAY, NIGHT, REST
int nRes = 4; // 4 resources: cost (0), #DAY (1), #NIGHT (2), #WORK (3)

//2- Create the schedule variables: the activity processed at each time slot
IntegerVariable[] sequence = makeIntVarArray("x",nTime,0,nAct-1,"cp:enum");
// - create the cost variables (one for each resource)
IntegerVariable[] bounds = new IntegerVariable[4];
bounds[0] = makeIntVar("z_0",30,80,"cp:bound"); // 30 <= cost <= 80
bounds[1] = makeIntVar("day",0,7,"cp:bound"); // 0 <= #DAY <= 7
bounds[2] = makeIntVar("night",3,7,"cp:bound"); // 3 <= #NIGHT <= 7
bounds[3] = makeIntVar("work",7,9,"cp:bound"); // 7 <= #WORK <= 9

//3- Create the automaton
Automaton auto = new Automaton();
// state 0: starting and accepting state
int start = auto.addState();
auto.setStartingState(start);
auto.setAcceptingState(start);
// state 1 and a transition (0,DAY,1)
int first = auto.addState();
auto.addTransition(start,first,DAY);
// state 2 and transitions (1,DAY,2), (1,NIGHT,2), (2,REST,0), (0,NIGHT,2)
int second = auto.addState();
auto.addTransition(first,second,new int[]{DAY,NIGHT});
auto.addTransition(second,start,REST);
auto.addTransition(start,second,NIGHT);
```

```
//4- Declare the assignment/transition costs:
// csts[i][j][s][r]: cost on resource r of assigning Xi to activity j at state s
int[][][][] csts = new int[nTime][nAct][auto.getNbStates()][nRes];
for (int i = 0 ; i < csts.length ; i++) {
    csts[i][DAY][0] = new int[]{3,1,0,1}; // costs of transition (0,DAY,1)
    csts[i][NIGHT][0] = new int[]{8,0,1,1}; // costs of transition (0,NIGHT,2)
    csts[i][DAY][1] = new int[]{5,1,0,1}; // costs of transition (1,DAY,2)
    csts[i][NIGHT][1] = new int[]{9,0,1,1}; // costs of transition (1,NIGHT,2)
    csts[i][REST][2] = new int[]{2,0,0,0}; // costs of transition (2,REST,0)
}

//5- Set a constraint parameter
MultiCostRegular.DATA_STRUCT = MultiCostRegular.LIST;

//6- add the constraint
m.addConstraint(multiCostRegular(sequence,bounds,auto,csts));

//7- create the solver, read the model and solve it
Solver s = new CPSolver();
s.read(m);
s.solve();
```

9.47 neq (constraint)

`neq` states that the two arguments are different:

$$x \neq y.$$

- **API** :
  - `neq(IntegerExpressionVariable x, IntegerExpressionVariable y)`
  - `neq(IntegerExpressionVariable x, int y)`
  - `neq(int x, IntegerExpressionVariable y)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete.*
- **references** :
  global constraint catalog: neq

Examples:

- example1:

```
Model m = new CPModel();
Solver s = new CPSolver();
int c = 1;
IntegerVariable v = makeIntVar("v", 0, 2);
m.addConstraint(neq(v, c));
s.read(m);
s.solve();
```

- example2

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 2);
IntegerVariable v2 = makeIntVar("v2", 0, 2);
IntegerExpressionVariable w1 = plus(v1, 1);
IntegerExpressionVariable w2 = minus(v2, 1);
m.addConstraint(neq(w1, w2));
s.read(m);
s.solve();
```

9.48 neqCard (constraint)

9.49 not (constraint)

`not`(c) holds if and only if constraint c does not hold:

$$\neg c$$

- **API** : `not(Constraint c)`
- **return type** : `Constraint`
- **options** : n/a
- **favorite domain** : n/a

Example :

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 10);
// x < 3
m.addConstraint(not(geq(x, 3)));

s.read(m);
s.solve();
```

9.50 notMember (constraint)

`notMember`(x, s) states that integer x is not contained in set s:

$$x \notin s$$

- **API** :
  - `notMember(int x, SetVariable s)`
  - `notMember(SetVariable s, int x)`
  - `notMember(SetVariable s, IntegerVariable x)`
  - `notMember(IntegerVariable x, SetVariable s)`

- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int x = 3;
int card = 2;
SetVariable y = makeSetVar("y", 2, 4);
m.addConstraint(notMember(y, x));
m.addConstraint(eqCard(y, card));
s.read(m);
s.solveAll();
```

9.51 nth (constraint)

`nth` is the well known *element* constraint. Several APIs are available:

- `nth`(i, x, y) ensures that $x[i] = y$
- `nth`(i, x, y, o) ensures that $x[i + o] = y$ (o is an *offset* for shifting values)
- `nth`(i, j, x, y) ensures that $x[i][j] = y$

- **API** :
  - `nth(IntegerVariable i, int[] x, IntegerVariable y)`
  - `nth(IntegerVariable i, IntegerVariable[] x, IntegerVariable y)`
  - `nth(IntegerVariable i, int[] x, IntegerVariable y, int offset)`
  - `nth(IntegerVariable i, IntegerVariable[] x, IntegerVariable y, int offset)`
  - `nth(IntegerVariable i, IntegerVariable j, int[][] x, IntegerVariable y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: element

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();

int[][] values = new int[][]{
    {1, 2, 0, 4, -323},
    {2, 1, 0, 3, 42},
    {6, 1, -7, 4, -40},
    {-1, 0, 6, 2, -33},
    {2, 3, 0, -1, 49}};
IntegerVariable index1 = makeIntVar("index1", -3, 10);
```

```
IntegerVariable index2 = makeIntVar("index2", -3, 10);
IntegerVariable var = makeIntVar("value", -20, 20);

m.addConstraint(nth(index1, index2, values, var));

s.read(m);
s.solveAll();
```

9.52 occurrence (constraint)

`occurrence`(v, z, x) states that z is equal to the number of elements in x with value v:

$$z = |\{i \mid x_i = v\}|$$

This is a specialization of the `globalCardinality` constraint.

- **API**: `occurrence(int v, IntegerVariable z, IntegerVariable... x)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: count

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();

IntegerVariable x1 = makeIntVar("X1", 0, 10);
IntegerVariable x2 = makeIntVar("X2", 0, 10);
IntegerVariable x3 = makeIntVar("X3", 0, 10);
IntegerVariable x4 = makeIntVar("X4", 0, 10);
IntegerVariable x5 = makeIntVar("X5", 0, 10);
IntegerVariable x6 = makeIntVar("X6", 0, 10);
IntegerVariable x7 = makeIntVar("X7", 0, 10);
IntegerVariable y1 = makeIntVar("Y1", 0, 10);

m.addConstraint(occurrence(3, y1, new IntegerVariable[]{x1, x2, x3, x4, x5, x6, x7}));

s.read(m);
s.solve();
```

9.53 occurrenceMax (constraint)

`occurrenceMax`(v, z, x) states that z is at most equal to the number of elements in x with value v:

$$z \leq |\{i \mid x_i = v\}|$$

This is a specialization of the `globalCardinality` constraint.

- **API**: `occurrenceMax(int v, IntegerVariable z, IntegerVariable... x)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: count

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();

IntegerVariable x1 = makeIntVar("X1", 0, 10);
IntegerVariable x2 = makeIntVar("X2", 0, 10);
IntegerVariable x3 = makeIntVar("X3", 0, 10);
IntegerVariable x4 = makeIntVar("X4", 0, 10);
IntegerVariable x5 = makeIntVar("X5", 0, 10);
IntegerVariable x6 = makeIntVar("X6", 0, 10);
IntegerVariable x7 = makeIntVar("X7", 0, 10);
IntegerVariable y1 = makeIntVar("Y1", 0, 10);

m.addConstraint(occurrenceMax(3, y1, new IntegerVariable[]{x1, x2, x3, x4, x5, x6, x7}));

s.read(m);
s.solve();
```

9.54 occurrenceMin (constraint)

`occurrenceMin`(v, z, x) states that z is at least equal to the number of elements in x with value v:

$$z \geq |\{i \mid x_i = v\}|$$

This is a specialization of the `globalCardinality` constraint.

- **API**: `occurrenceMin(int v, IntegerVariable z, IntegerVariable... x)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: count

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();

IntegerVariable x1 = makeIntVar("X1", 0, 10);
IntegerVariable x2 = makeIntVar("X2", 0, 10);
IntegerVariable x3 = makeIntVar("X3", 0, 10);
IntegerVariable x4 = makeIntVar("X4", 0, 10);
IntegerVariable x5 = makeIntVar("X5", 0, 10);
```

```
IntegerVariable x6 = makeIntVar("X6", 0, 10);
IntegerVariable x7 = makeIntVar("X7", 0, 10);
IntegerVariable y1 = makeIntVar("Y1", 0, 10);

m.addConstraint(occurrenceMin(3, y1, new IntegerVariable[]{x1, x2, x3, x4, x5, x6, x7}));

s.read(m);
s.solve();
```

9.55 oppositeSign (constraint)

verify case 0

oppositeSign(x, y) states that the two arguments have opposite signs:

$$xy \leq 0$$

- **API** : `oppositeSign(IntegerExpressionVariable x, IntegerExpressionVariable y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", -1, 1);
IntegerVariable y = makeIntVar("y", -1, 1);
IntegerVariable z = makeIntVar("z", 0, 1000);
m.addConstraint(oppositeSign(x,y));
m.addConstraint(eq(z, plus(mult(x, -425), mult(y, 391))));
s.read(m);
s.solve();
System.out.println(s.getVar(z).getVal());
```

9.56 or (constraint)

or$(c_1, \ldots, c_n)$ states that one or more of the constraints in arguments are satisfied:

$$c_1 \vee c_2 \vee \ldots \vee c_n$$

- **API** : `or(Constraint... c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();

IntegerVariable v1 = makeIntVar("v1", 0, 1);
IntegerVariable v2 = makeIntVar("v2", 0, 1);

m.addConstraint(or(eq(v1, 1), eq(v2, 1)));

s.read(m);
s.solve();
```

9.57 pack (constraint)

`pack(items, load, bin, size)` states that a collection of items is packed into different bins, such that the total size of the items in each bin does not exceed the bin capacity:

$$\texttt{load}[b] = \sum_{i \in \texttt{items}[b]} \texttt{size}[i], \quad \forall \text{ bin } b$$

$$i \in \texttt{items}[b] \iff \texttt{bin}[i] = b, \quad \forall \text{ bin } b, \ \forall \text{ item } i$$

`pack` is a bin packing constraint based on [**?**].

- **API** :
  - `pack(SetVariable[] items, IntegerVariable[] load, IntegerVariable[] bin, IntegerConstantVariable [] size, String... options)`
  - `pack(PackModeler modeler,String... options)`: PackModeler is a high-level modeling object.
  - `pack(int[] sizes, int nbBins, int capacity, String... options)`: build instance with PackModeler.
- **Variables**:
  - `SetVariable[] items`: `items`$[b]$ is the set of items packed into bin b.
  - `IntegerVariable[] load`: `load`$[b]$ is the total size of the items packed into bin b.
  - `IntegerVariable[] bin`: `bin`$[i]$ is the bin where item i is packed into.
  - `IntegerConstantVariable[] size`: `size`$[i]$ is the size of item i.
- **return type** : `Constraint`
- **options** :
  - `SettingType.ADDITIONAL_RULES`: additional filtering rules *recommended*
  - `SettingType.DYNAMIC_LB`: feasibility tests based on dynamic lower bounds for 1D-bin packing
- **favorite domain** : *to complete*
- **references** :
  - [**?**]: *A constraint for bin packing*
  - global constraint catalog: bin_packing (variant)

Example:
Take a look at *choco.samples.pack* to see advanced use of the constraint.

```
Model m = new CPModel();
m.addConstraint(Choco.pack(new int[]{5,3,2,6,8,5},5,10, SettingType.ADDITIONAL_RULES.
    getOptionName()));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

9.58 precedenceReified (constraint)

`precedenceReified`(x_1, d, x_2, b) states that x_1 plus duration d is less than or equal to x_2 requires boolean b to be true:

$$b \iff x_1 + d \leq x_2$$

- **API** : `precedenceReified(IntegerVariable x1, int d, IntegerVariable x2, IntegerVariable b)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example: `//TODO: complete`

9.59 preceding (constraint)

verify the spec when one duration is null
`preceding`(x_1, d_1, x_2, d_2, b) states the precedence order between two disjunctive tasks defined by a starting time x and a duration d: If task 1 precedes task 2, then b is true, otherwise if task 2 precedes task 1, then b is false, otherwise the constraint does not hold:

$$(b \wedge (x_1 + d_1 \leq x_2)) \quad \vee \quad (\neg b \wedge (x_2 + d_2 \leq x_1))$$

- **API** : `preceding(IntegerVariable x1, int d1, IntegerVariable x2, int d2, IntegerVariable b)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example: `//TODO: complete`

9.60 regular (constraint)

`regular`$(x, \mathcal{L}(\Pi))$ states that sequence x is a word belonging to the regular language $\mathcal{L}(\Pi)$:

$$(x_1, \ldots, x_n) \in \mathcal{L}(\Pi)$$

The accepting language can be specified either by a deterministic finite automaton (DFA), a list of feasible or infeasible tuples, or a regular expression:

DFA: Automaton Π is defined on a given *alphabet* $\Sigma \subseteq \mathbb{Z}$ by a set $Q = \{0, \ldots, m\}$ of *states*, a subset $A \subseteq Q$ of *final* or *accepting states* and a table $\Delta \subseteq Q \times \Sigma \times Q$ of *transitions* between states. Δ is encoded as `List<Transition>` where a Transition object $\delta =$ `new Transition`(q_i, σ, q_j) is made of three integers expressing the ingoing state q_i, the label σ, and the outgoing state q_j. Automaton Π is a DFA if Δ is finite and if it has only one initial state (here, state 0 is considered as the unique initial state) and no two transitions sharing the same ingoing state and the same label.

feasible tuples: *regular* can be used as an extensional constraint. Given the list of *feasible* tuples for sequence x, this API builds a DFA from the list, and then enforces GAC on the constraint. Using `regular` can be more efficient than a standard GAC algorithm on tables of tuples if the tuples are structured so that the resulting DFA is compact. The DFA is built from the list of tuples by computing incrementally the minimal DFA after each addition of tuple.

infeasible tuples: An another API allows to specify the list of *infeasible* tuples and then builds the corresponding feasible DFA. This operation requires to know the entire alphabet, hence this API has two mandatory table fields *min* and *max* defining the minimum and maximum values of each variable x_i.

regular expression: Finally, the `regular` constraint can be based on a regular expression, such as `String regexp = "(12)(3*)";`— This expression recognizes any (possibly empty) sequences of 3 preceded by at least one 1 or one 2.

- **API** :
  - `regular(DFA pi, IntegerVariable[] x)`
  - `regular(IntegerVariable[] x, List<int[]> feasTuples)`
  - `regular(IntegerVariable[] x, List<int[]> infeasTuples, int[] min, int[] max)`
  - `regular(String regexp, IntegerVariable[] x)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  [?]: *A regular language membership constraint*

Examples:

- example 1 with DFA:

```
//1- Create the model
Model m = new CPModel();
int n = 6;
IntegerVariable[] vars = new IntegerVariable[n];
for (int i = 0; i < vars.length; i++) {
    vars[i] = makeIntVar("v" + i, 0, 5);
}
//2- Build the list of transitions of the DFA
List<Transition> t = new LinkedList<Transition>();
t.add(new Transition(0, 1, 1));
t.add(new Transition(1, 1, 2));
// transition with label 1 from state 2 to state 3
t.add(new Transition(2, 1, 3));
t.add(new Transition(3, 3, 0));
```

```
t.add(new Transition(0, 3, 0));

//3- Two final states: 0, 3
List<Integer> fs = new LinkedList<Integer>();
fs.add(0); fs.add(3);

//4- Build the DFA
DFA auto = new DFA(t, fs, n);

//5- add the constraint
m.addConstraint(regular(auto, vars));

//6- create the solver, read the model and solve it
Solver s = new CPSolver();
s.read(m);
s.solve();
do {
    for (int i = 0; i < n; i++)
    System.out.print(s.getVar(vars[i]).getVal());
    System.out.println("");
} while (s.nextSolution());

//7- Print the number of solution found
System.out.println("Nb_sol␣:␣" + s.getNbSolutions());
```

- example 2 with feasible tuples:

```
//1- Create the model
Model m = new CPModel();
IntegerVariable v1 = makeIntVar("v1", 1, 4);
IntegerVariable v2 = makeIntVar("v2", 1, 4);
IntegerVariable v3 = makeIntVar("v3", 1, 4);

//2- add some allowed tuples (here, the tuples define a all_equal constraint)
List<int[]> tuples = new LinkedList<int[]>();
tuples.add(new int[]{1, 1, 1});
tuples.add(new int[]{2, 2, 2});
tuples.add(new int[]{3, 3, 3});
tuples.add(new int[]{4, 4, 4});

//3- add the constraint
m.addConstraint(regular(new IntegerVariable[]{v1, v2, v3}, tuples));

//4- Create the solver, read the model and solve it
Solver s = new CPSolver();
s.read(m);
s.solve();
do {
    System.out.println("("+s.getVar(v1)+","+s.getVar(v2)+","+s.getVar(v3)+")");
} while (s.nextSolution());

//5- Print the number of solution found
System.out.println("Nb_sol␣:␣" + s.getNbSolutions());
```

- example 3 with regular expression:

```
//1- Create the model
Model m = new CPModel();
int n = 6;
```

```
IntegerVariable[] vars = makeIntVarArray("v", n, 0, 5);

//2- add the constraint
String regexp = "(1|2)(3*)(4|5)";
m.addConstraint(regular(regexp, vars));

//3- Create the solver, read the model and solve it
Solver s = new CPSolver();
s.read(m);
s.solve();
do {
    for (int i = 0; i < n; i++)
        System.out.print(s.getVar(vars[i]).getVal());
    System.out.println("");
} while (s.nextSolution());

//4- Print the number of solution found
System.out.println("Nb_sol␣:␣" + s.getNbSolutions());
```

9.61 reifiedIntConstraint (constraint)

- `reifiedIntConstraint`(b, c) states that boolean b is true if and only if constraint c holds:

$$b \iff c$$

- `reifiedIntConstraint`(b, c_1, c_2) states that boolean b is true if and only if c_1 holds, and b is false if and only if c_2 holds (c_2 must be the opposite constraint of c_1):

$$(b \wedge c_1) \vee (\neg b \wedge c_2)$$

- **API** :
  - `reifiedIntConstraint(IntegerVariable b, Constraint c)`
  - `reifiedIntConstraint(IntegerVariable b, Constraint c1, Constraint c2)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

Parameter b is a boolean variable (enumerated domain with two values $\{0, 1\}$) and c is a constraint over Integer variables.

The constraint c to reify has to provide its opposite (the opposite is needed for propagation). Most basic constraints of Choco provides their opposite by default, and can then be reified using the first API. The second API attends to reify user-defined constraints as it allows the user to directly specify the opposite constraint.

Example:

```
CPModel m = new CPModel();
CPSolver s = new CPSolver();

IntegerVariable b = makeIntVar("b", 0, 1);
IntegerVariable x = makeIntVar("x", 0, 10);
IntegerVariable y = makeIntVar("y", 0, 10);
```

```
// reified constraint (x<=y)
m.addConstraint(reifiedIntConstraint(b, leq(x, y)));

s.read(m);
s.solveAll();
```

9.62 relationPairAC (constraint)

`relationPairAC`(x, y, rel) states an extensional binary constraint on (x, y) defined by the binary relation rel:

$$(x, y) \in rel$$

Many constraints of the same kind often appear in a model. Relations can therefore often be shared among many constraints to spare memory.

The API is duplicated to allow definition of options.

- **API** :
  - `relationPairAC(IntegerVariable x, IntegerVariable y, BinRelation rel)`
  - `relationPairAC(String options, IntegerVariable x, IntegerVariable y, BinRelation rel)`
- **return type** : `Constraint`
- **options** :
  - *no option* : use AC3 (default arc consistency)
  - `cp:ac3`: to get AC3 algorithm (searching from scratch for supports on all values)
  - `cp:ac2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `cp:ac32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `cp:ac322`: to get AC3 with the used of `BitSet` to know if a support still exists
- **favorite domain** : *to complete*

Example:

```
private class MyEquality extends CouplesTest {

    public boolean checkCouple(int x, int y) {
    return x == y;
    }

    public void example(){
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable v1 = makeIntVar("v1", 1, 4);
        IntegerVariable v2 = makeIntVar("v2", 1, 4);
        IntegerVariable v3 = makeIntVar("v3", 3, 6);
        m.addConstraint(relationPairAC("cp:ac32",v1, v2, new MyEquality()));
        m.addConstraint(relationPairAC("cp:ac32",v2, v3, new MyEquality()));
        s.read(m);
        s.solveAll();
    }
}
```

9.63 relationTupleAC (constraint)

`relationTupleAC`(x, rel) states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the n-ary relation rel, and then enforces arc consistency:

$$(x_1, \ldots, x_n) \in rel$$

Many constraints of the same kind often appear in a model. Relations can therefore often be shared among many constraints to spare memory. The API is duplicated to define options.

- **API**:
  - `relationTupleAC(IntegerVariable[] x, LargeRelation rel)`
  - `relationTupleAC(String options, IntegerVariable[] x, LargeRelation rel)`
- **return type**: `Constraint`
- **options** :
  - *no option*: use AC32 (default arc consistency)
  - `cp:ac32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `cp:ac2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `cp:ac2008`: to get AC2008 algorithm (maintained by STR)
- **favorite domain** : *to complete*

Example : `//TODO : add example`

9.64 relationTupleFC (constraint)

`relationTupleFC`(x, rel) states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the n-ary relation rel, and then enforces forward checking:

$$(x_1, \ldots, x_n) \in rel$$

Many constraints of the same kind often appear in a model. Relations can therefore often be shared among many constraints to spare memory.

- **API**: `relationTupleFC(IntegerVariable[] x, LargeRelation rel)`
- **return type**: `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*

Example : `//TODO : add example`

9.65 sameSign (constraint)

verify case 0

`sameSign`(x, y) states that the two arguments have the same sign:

$$xy \geq 0$$

- **API** : `sameSign(IntegerExpressionVariable x, IntegerExpressionVariable y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", -1, 1);
IntegerVariable y = makeIntVar("y", -1, 1);
IntegerVariable z = makeIntVar("z", 0, 1000);
m.addConstraint(oppositeSign(x,y));
m.addConstraint(eq(z, plus(mult(x, -425), mult(y, 391))));
s.read(m);
s.solve();
System.out.println(s.getVar(z).getVal());
```

9.66 setDisjoint (constraint)

`setDisjoint`(s_1, s_2) states that the two set arguments are disjoint:

$$s_1 \cap s_2 = \emptyset$$

- **API** : `setDisjoint(SetVariable s1, SetVariable s2)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVar x = makeSetVar("X", 1, 3);
SetVar y = makeSetVar("Y", 1, 3);
Constraint c1 = setDisjoint(x, y);
m.addConstraint(c1);
s.read(m);
s.solveAll();
```

9.67 setInter (constraint)

`setInter`(s_1, s_2, s_3) states that the third set s_3 is exactly the intersection of the two first sets:

$$s_1 \cap s_2 = s_3$$

- **API** : `setInter(SetVariable s1, SetVariable s2, SetVariable s3)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
setVar x = makeSetVar("X", 1, 3);
SetVar y = makeSetVar("Y", 1, 3);
SetVar z = makeSetVar("Z", 2, 3);
Constraint c1 = setInter(x, y, z);
m.addConstraint(c1);
s.read(m);
s.solveAll();
```

9.68 setUnion (constraint)

`setUnion`(s_1, s_2, s_3) states that the third set s_3 is exactly the union of the two first sets:

$$s_1 \cup s_2 = s_3$$

- **API** : `setUnion(SetVariable s1, SetVariable s2, SetVariable s3)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
setVar x = makeSetVar("X", 1, 3);
SetVar y = makeSetVar("Y", 3, 5);
SetVar z = makeSetVar("Z", 0, 6);
Constraint c1 = setUnion(x, y, z);
m.addConstraint(c1);
s.read(m);
s.solveAll();
```

9.69 sorting (constraint)

9.70 stretchCyclic (constraint)

stretchCyclic states that ... *to complete.*

- **API** :
- **return type** : Constraint
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

9.71 stretchPath (constraint)

A *stretch* in a sequence x is a maximum subsequence of (consecutive) identical values. `stretchPath`$(param, x)$ enforces the minimal and maximal length of the stretches in sequence x of any values given in *param*: Consider the sequence x as a concatenation of stretches $x^1.x^2 \dots x^k$ with v^i and l^i being respectively the value and the length of stretch x^i,

$$\forall i \in \{1, \dots, k\},\ \forall j, \quad param[j][0] = v^i \quad \Longrightarrow \quad param[j][1] \leq l^i \leq param[j][2]$$

Useful for Rostering Problems. `stretchPath` is implemented by a `regular` constraint that performs GAC. The bounds on the stretch lengths are defined by *param* a list of triples of integers: $[value, min, max]$ specifying the minimal and maximal lengths of any stretch of the corresponding value.

This API requires a Java library on automaton available on http://www.brics.dk/automaton/. (It is contained in the Choco jar file.)

- **API** : `stretchPath(List<int[]> param, IntegerVariable... x)`
- **return type** : Constraint
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  - [?]: *A regular language membership constraint*
  - global constraint catalog: stretch_path

Example:

```
Model m = new CPModel();
int n = 7;
IntegerVariable[] vars = makeIntVarArray("v", n, 0, 2);

//define the stretches
ArrayList<int[]> lgt = new ArrayList<int[]>();
lgt.add(new int[]{0, 2, 2}); // stretches of value 0 are of length 2
lgt.add(new int[]{1, 2, 3}); // stretches of value 1 are of length 2 or 3
lgt.add(new int[]{2, 2, 2}); // stretches of value 2 are of length 2

m.addConstraint(stretchPath(lgt, vars));
```

```
Solver s = new CPSolver();
s.read(m);
s.solve();
```

9.72 times (constraint)

`times`(x_1, x_2, x_3) states that the third argument is equal to the product of the two arguments:

$$x_3 = x_1 \times x_2.$$

- **API**:
  - `times(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3)`
  - `times(int x1, IntegerVariable x2, IntegerVariable x3)`
  - `times(IntegerVariable x1, int x2, IntegerVariable x3)`
- **return type** : `Constraint`
- **option** : *n/a*
- **favorite domain**: bound

Example:

```
Model m = new CPModel();
IntegerVariable x = makeIntVar("x", 1, 2);
IntegerVariable y = makeIntVar("y", 3, 5);
IntegerVariable z = makeIntVar("z", 3, 10);
m.addConstraint(times(x, y, z));
s.setVarIntSelector(new RandomIntVarSelector(s, i));
s.setValIntSelector(new RandomIntValSelector(i + 1));
s.read(m);
s.solve();
```

9.73 tree (constraint)

Let $G = (V, A)$ be a digraph on $V = \{1, \ldots, n\}$. G can be modeled by a sequence of domain variables $x = (x_1, \ldots, x_n) \in V^n$ – the *successors* variables – whose respective domains are given by $D_i = \{j \in V \mid (i, j) \in A\}$. Conversely, when instantiated, x defines a subgraph $G_x = (V, A_x)$ of G with $A_x = \{(i, x_i) \mid i \in V\} \subseteq A$. Such a subgraph has one particularity: any connected component of G_x contains either no loop – and then it contains a cycle – or exactly one loop $x_i = i$ and then it is a *tree* of root i (literally, it is an anti-arborescence as there exists a path from each node to i and i has a loop).

`tree`$(x, restrictions)$ is a vertex-disjoint graph partitioning constraint. It states that G_x is a forest (its connected components are trees) that satisfies some conditions specified by *restrictions*. `tree` deals with several kinds of graph restrictions on:

- the number of trees
- the number of proper trees (a tree is proper if it contains more than 2 nodes)
- the weigth of the partition: the sum of the weights of the edges
- incomparability: some nodes in pairs have to belong to distinct trees
- precedence: some nodes in pairs have to belong to the same tree in a given order
- conditional precedence: some nodes in pairs have to respect a given order if they belong to the same tree
- the in-degree of the nodes
- the time windows on nodes (given travelling times on arcs)

Many applications require to partition a graph such that each component contains exactly one *resource* node and several *task* nodes. A typical example is a routing problem where vehicle routes are paths (a path is a special case of tree) starting from a depot and delivering goods to several clients. Another example is a local network where each computer has to be connected to one shared printer. Last, one can cite the problem of reconstructing plylogeny trees. The constraint `tree` can handle these kinds of problems with many additional constraints on the structure of the partition.

- **API** : `tree(TreeParametersObject param)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  - [**?**]: *Combining tree partitioning, precedence, and incomparability constraints*
  - global constraint catalog: proper_forest (variant)

The tree constraint API requires a particular Model object, named `TreeParametersObject`. It can be created with the following parameters:

parameter	type	description
n	`int`	number of nodes in the initial graph G
nTree	`IntegerVariable`	number of trees in the resulting forest G_x
nProper	`IntegerVariable`	number of proper trees in G_x
objective	`IntegerVariable`	(bounded) total cost of G_x
graphs	`List<BitSet[]>`	graphs encoded as successor lists, `graphs[0]` the initial graph G, `graphs[1]` a precedence graph, `graphs[2]` a conditional precedence graph, `graphs[3]` an incomparability graph
matrix	`List<int[][]>`	`matrix[0]` the indegree of each node, and `matrix[1]` the starting time from each node
travel	`int[][]`	the travel time of each arc

Example:

```
Model m = new CPModel();
int nbNodes = 7;

//1- create the variables involved in the partitioning problem
IntegerVariable ntree = makeIntVar("ntree",1,5);
IntegerVariable nproper = makeIntVar("nproper",1,1);
IntegerVariable objective = makeIntVar("objective",1,100);

//2- create the different graphs modeling restrictions
List<BitSet[]> graphs = new ArrayList<BitSet[]>();
BitSet[] succ = new BitSet[nbNodes];
BitSet[] prec = new BitSet[nbNodes];
BitSet[] condPrecs = new BitSet[nbNodes];
BitSet[] inc = new BitSet[nbNodes];
for (int i = 0; i < nbNodes; i++) {
    succ[i] = new BitSet(nbNodes);
    prec[i] = new BitSet(nbNodes);
    condPrecs[i] = new BitSet(nbNodes);
    inc[i] = new BitSet(nbNodes);
}

 // initial graph (encoded as successors variables)
succ[0].set(0,true); succ[0].set(2,true); succ[0].set(4,true);
succ[1].set(0,true); succ[1].set(1,true); succ[1].set(3,true);
succ[2].set(0,true); succ[2].set(1,true); succ[2].set(3,true); succ[2].set(4,true);
succ[3].set(2,true); succ[3].set(4,true); // successor of 3 is either 2 or 4
succ[4].set(2,true); succ[4].set(3,true);
succ[5].set(4,true); succ[5].set(5,true); succ[5].set(6,true);
succ[6].set(3,true); succ[6].set(4,true); succ[6].set(5,true);

 // restriction on precedences
prec[0].set(4,true); // 0 has to precede 4
prec[4].set(3,true); prec[4].set(2,true);
prec[6].set(4,true);

 // restriction on conditional precedences
condPrecs[5].set(1,true); // 5 has to precede 1 if they belong to the same tree

 // restriction on incomparability:
inc[0].set(6,true); inc[6].set(0,true); // 0 and 6 have to belong to distinct trees

graphs.add(succ);
graphs.add(prec);
graphs.add(condPrecs);
graphs.add(inc);

//3- create the different matrix modeling restrictions
List<int[][]> matrix = new ArrayList<int[][]>();

 // restriction on bounds on the indegree of each node
int[][] degree = new int[nbNodes][2];
for (int i = 0; i < nbNodes; i++) {
    degree[i][0] = 0; degree[i][1] = 2; // 0 <= indegree[i] <= 2
}
matrix.add(degree);

 // restriction on bounds on the starting time at each node
int[][] tw = new int[nbNodes][2];
for (int i = 0; i < nbNodes; i++) {
    tw[i][0] = 0; tw[i][1] = 100; // 0 <= start[i] <= 100
```

```
}
tw[0][1] = 15;      // 0 <= start[0] <= 15
tw[2][0] = 35; tw[2][1] = 40;  // 35 <= start[2] <= 45
tw[6][1] = 5;      // 0 <= start[6] <= 5
matrix.add(tw);

//4- matrix for the travel time between each pair of nodes
int[][] travel = new int[nbNodes][nbNodes];
for (int i = 0; i < nbNodes; i++) {
    for (int j = 0; j < nbNodes; j++) travel[i][j] = 100000;
}
travel[0][0] = 0; travel[0][2] = 10; travel[0][4] = 20;
travel[1][0] = 20; travel[1][1] = 0; travel[1][3] = 20;
travel[2][0] = 10; travel[2][1] = 10; travel[2][3] = 5; travel[2][4] = 5;
travel[3][2] = 5; travel[3][4] = 2;
travel[4][2] = 5; travel[4][3] = 2;
travel[5][4] = 15; travel[5][5] = 0; travel[5][6] = 10;
travel[6][3] = 5; travel[6][4] = 20; travel[6][5] = 10;

//5- create the input structure and the tree constraint
TreeParametersObject parameters = new TreeParametersObject(nbNodes, ntree, nproper, objective
    , graphs, matrix, travel);
Constraint c = Choco.tree(parameters);

m.addConstraint(c);
Solver s = new CPSolver();
s.read(m);

//6- heuristic: choose successor variables as the only decision variables
s.setVarIntSelector(new StaticVarOrder(s.getVar(parameters.getSuccVars())));
CPSolver.setVerbosity(CPSolver.SOLUTION);
s.solveAll();
```

9.74 TRUE (constraint)

TRUE always returns *true*.

Part III

Tutorials

If you look for an easy step-by-step program in CHOCO, getting started is for you! It introduces to basic concepts of a CHOCO program (Model, Solver, variables and constraints...).

This part also presents a collection of exercises with their solutions. It covers simple to advanced uses of CHOCO.

See also old pages: `http://choco.sourceforge.net/tut_expl.html`

Chapter 10

Getting started: welcome to Choco

This introduction covers the basics of writing a program in Choco

Choco is a java library for constraint satisfaction problems (CSP), constraint programming (CP) and explanation-based constraint solving (e-CP). It is built on a event-based propagation mechanism with backtrackable structures.

10.1 Before starting

Before doing anything, you have to be sure that

- you have at least Java6 installed on your environment.
- you have a IDE (like IntelliJ IDEA or Eclipse).

To install Java6 or your IDE, please refer to its specific documentation. We now assume that you have the previously defined environment.

You need to create a **New Project...** on your favorite IDE (create a new project on IntelliJ, create a new project on Eclipse). Our project name is *ChocoProgram*. Create a new class, named *MyFirstChocoProgram*, with a main method.

```
public class MyFirstChocoProgram {

    public static void main(String[] args) {

    }
}
```

10.2 Download Choco

Now, before doing anything else, you need to download the last stable version of Choco. See the download page. Once you have download choco-2.0.0.*X*.jar (*X* is the last stable version indicator), you need to add it to the classpath of your project. (see the faq for more informations).

Now you are ready to create you first Choco program.

If you want a short introduction on what is constraint programming, you can find some informations on the introduction. If you prefer start with an example, please refer to introduction#my first choco program. When you feel ready, solve your own problem! And if you need more tries, please take a look at the exercises.

Chapter 11

First Example: Magic square

A simple magic square of order 3 can be seen as the "Hello world!" program in Choco. First of all, we need to agree on the definition of a magic square of order 3. Wikipedia tells us that :

> A **magic square** of order n is an arrangement of n^2 numbers, usually distinct integers, in a square, such that the n numbers in all rows, all columns, and both diagonals sum to the same constant. A normal magic square contains the integers from 1 to n^2.

So we are going to solve a problem where unknows are cells value, knowing that each cell can take its value between 1 and n^2, is different from the others and columns, diagonals and rows are equal to the same constant M (which is equal to $n * (n^2 + 1)/2$).

We have the definition, let see how to add some Choco in it.

11.1 First, the model

To define our problem, we need to create a Model object. As we want to solve our problem with constraint programming (of course, we do), we need to create a CPModel.

```
//constants of the problem:
int n = 3;
int M = n*(n*n+1)/2;

// Our model
Model m = new CPModel();
```

These objects require to import the following classes:

```
import choco.cp.model.CPModel;
import choco.kernel.model.Model;
```

At the begining, our model is empty, no problem has been defined explicitly. A model is composed of variables and constraints, and constraints link variables to each others.

- **Variables** A variable is an object defined by a name, a type and a domain. We know that our unknowns are cells of the magic square. So: `IntegerVariable cell = Choco.makeIntVar("aCell", 1, n*n);` which means that *aCell* is an integer variable, and its domain is defined from *1* to *n*n*. But we need n^2 variables, so the easiest way to define them is:

```
IntegerVariable[][] cells = new IntegerVariable[n][n];
for(int i = 0; i < n; i++){
   for(int j = 0; j < n; j++){
      cells[i][j] = Choco.makeIntVar("cell"+j, 1, n*n);
      m.addVariables(cells[i][j]);
```

```
    }
  }
```

This code requires to import the following classes:

```
import choco.kernel.model.variables.integer.IntegerVariable;
import choco.Choco;
```

We add each variables to our model: `m.addVariables(cells[i][j]);`
Now that our variables are defined, we have to define the constraints between variables.

- **Constraints over the rows** The sum of ach rows is equal to a constant M. So we need a sum operator and and equality constraint. The both are provides by the `Choco.java` class.

```
//Constraints
// ... over rows
Constraint[] rows = new Constraint[n];
for(int i = 0; i < n; i++){
   rows[i] = Choco.eq(Choco.sum(cells[i]), M);
}
```

This part of code requires the following import:

```
import choco.kernel.model.constraints.Constraint;
```

After the creation of the constraints, we need to add them to the model:

```
m.addConstraints(rows);
```

- **Constraints over the columns** Now, we need to declare the equality between the sum of each column and M. But, the way we have declare our variables matrix does not allow us to deal easily with it in the column case. So we create the transposed matrix (a 90^o rotation of the matrix) of *cells*.

> We do not introduce new variables. We just reorder the matrix to see the *column point of view*.

```
//... over columns
// first, get the columns, with a temporary array
IntegerVariable[][] cellsDual = new IntegerVariable[n][n];
for(int i = 0; i < n; i++){
   for(int j = 0; j < n; j++){
      cellsDual[i][j] = cells[j][i];
   }
}
```

Now, we can declare the constraints as before:

```
Constraint[] cols = new Constraint[n];
for(int i = 0; i < n; i++){
   cols[i] = Choco.eq(Choco.sum(cellsDual[i]), M);
}
```

And we add them to the model:

```
m.addConstraints(cols);
```

- **Constraints over the diagonals** Now, we get the two diagonals array *diags*, reordering the required *cells* variables, like in the previous step.

```
//... over diagonals
IntegerVariable[][] diags = new IntegerVariable[2][n];
for(int i = 0; i < n; i++){
    diags[0][i] = cells[i][i];
    diags[1][i] = cells[i][(n-1)-i];
}
```

And we add the constraints to the model (in one step this time).

```
m.addConstraint(Choco.eq(Choco.sum(diags[0]), M));
m.addConstraint(Choco.eq(Choco.sum(diags[1]), M));
```

- **Constraints of variables AllDifferent** Finally, we add the AllDifferent constraints, stating that each *cells* variables takes a unique value. One more time, we have to reorder the variables, introducing temporary array.

```
//All cells are differents from each other
IntegerVariable[] allVars = new IntegerVariable[n*n];
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        allVars[i*n+j] = cells[i][j];
    }
}
m.addConstraint(Choco.allDifferent(allVars));
```

11.2 Then, the solver

Our model is established, it does not require any other information, we can focus on the way to solve it. The first step is to create a Solver;

```
//Our solver
Solver s = new CPSolver();
```

This part requires the following imports:

```
import choco.kernel.solver.Solver;
import choco.cp.solver.CPSolver;
```

After that, the model and the solver have to be linked, thus the solver *read* the model, to extract informations:

```
//read the model
s.read(m);
```

Once it is done, we just need to solve it:

```
//solve the problem
s.solve();
```

And print the information

```
//Print the values
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        System.out.print(s.getVar(cells[i][j]).getVal()+" ");
    }
    System.out.println();
}
```

11.3 Conclusion

We have seen, in a few steps, how to solve a basic problem using constraint programming and Choco. Now, you are ready to solve your own problem, and if you need more tries, please take a look at the exercises. You can download the java file of the introduction: myfirstchocoprogram.zip

Chapter 12

Exercises

12.1 I'm new to CP

The goal of this practical work is twofold :

- **problem modelling** with the help of variables and constraints ;
- **mastering the syntax** of Choco in order to tackle basic problems.

Warning, constraint modelling should not be solver specific. That is why you are strongly advised to write down your model before starting its implementation within Choco.

12.1.1 Exercise 1.1 (A soft start)

Algorithm 1 (below) describes a problem which fits the minimum choco syntax requirements.

Question 1 describe the constraint network modelled in Algorithm 1.

Question 2 give the variable domains after constraint propagation.

Algorithm 1 Mysterious model.

```
// Build a model
Model m = new CPModel() ;

// Build enumerated domain variables
IntegerVariable x1 = makeIntVar("var1", 0, 5);
IntegerVariable x2 = makeIntVar("var2", 0, 5);
IntegerVariable x3 = makeIntVar("var3", 0, 5);

// Build the constraints
Constraint C1 = gt(x1, x2) ;
Constraint C2 = neq(x1, x3) ;
Constraint C3 = gt(x2, x3) ;

// Add the constraints to the Choco model
m.addConstraint(C1) ;
m.addConstraint(C2) ;
m.addConstraint(C3) ;

// Build a solver
Solver s = new CPSolver();
```

```
// Read the model
s.read(m);

// Solve the problem
s.solve() ;

// Print the variable domains
System.out.println("var1␣=" + s.getVar(x1) .getVal()) ;
System.out.println("var2␣=" + s.getVar(x2) .getVal()) ;
System.out.println("var3␣=" + s.getVar(x3) .getVal()) ;
```

(Solution)

12.1.2 Exercise 1.2 (DONALD + GERALD = ROBERT)

Associate a different digit to every letter so that the equation DONALD + GERALD = ROBERT is verified.

(Solution)

12.1.3 Exercise 1.3 (A famous example. . . a sudoku grid)

A sudoku grid is a square composed of nine squares called *blocks*. Each block is itself composed of 3x3 cells (see figure 1). The purpose of the game is to fill the grid so that each block, column and row contains all the numbers from 1 to 9 once and only once

Question 1 propose a way to model the sudoku problem with difference constraints. Implement your model with Choco.

Question 2 which global constraint can be used to model such a problem ? Modify your code accordingly.

Question 3 Test, for both models, the initial propagation step (use Choco `propagate()` method). What can be noticed ? What is the point in using global constraints ?

		7	5			3		
	4			2		1		
1				7			5	
		3	1	4		2		6
4				6	2	7		
	6	5		3				8
	7	1				6		
8								
	5		7				4	1

Figure 12.1: An exemple of a Sudoku grid

(Solution)

12.1.4 Exercise 1.4 (The knapsack problem)

Let us organise a trek. Each hiker carries a knapsack of capacity 34 and can store 3 kinds of food which respectively supply energetic values (6,4,2) for a consumed capacity of (7,5,3). The problem is to find which food is to be put in the knapsack so that the energetic value is maximal.

Question 1 In the first place, we will not consider the idea of maximizing the energetic value. Try to find a satisfying solution by modelling and implementing the problem within choco.

Question 2 Find and use the choco method to **maximise** the energetic value of the knapsack.

Question 3 Propose a Value selector heuristic to improve the efficiency of the model.

(Solution)

12.1.5 Exercise 1.5 (The n-queens problem)

The n-queens problem aims to place n queens on a chessboard of size n so that no queen can attack one another.

Question 1 propose and implement a model based on one L_i variable for every row. The value of L_i indicates the column where a queen is to be put. Use simple difference constraints and confirm that 92 solutions are obtained for $n = 8$.

Question 2 Add a redundant model by considering variables on the columns (C_i). Continue to use simple difference constraints.

Question 3 Compare the number of nodes created to find the solutions with both models. How can you explain such a difference ?

Question 4 Add to the previously implemented model the following heuristics:

- Select first the line variable L_I which has the smallest domain ;
- Select the value $j \in L_i$ so that the associated column variable C_j has the smallest domain.

Again, compare both approaches in term of number of nodes and solving time to find ONE solution for $n = 75, 90, 95, 105$.

Question 5 what changes are caused by the use of the global constraint `alldifferent` ?

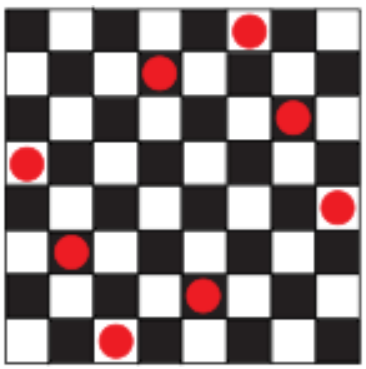

Figure 12.2: A solution of the n-queens problem for $n = 8$

(Solution)

12.2 I know CP

12.2.1 Exercise 2.1 (Bin packing, cumulative and search strategies)

Can n objects of a given size fit in m bins of capacity C ? The problem is here stated has a satisfaction problem for the sake of simplicity. Your model and heuristics will be checked by generating random instances for given n and C. The random generation must be reproducible.

Question 1 Propose a boolean model (0/1 variables).

Question 2 Let us turn this satisfaction problem into an optimization one. Use your previously stated model but increase regularly the number of containers until a feasible solution is found.

Question 3 Implement a naive lower bound. This can be done by considering the occupied size globally.

Question 4 Propose a model with integer variables based on the cumulative constraint (see choco user guide/API for details). Define an objective function to minimize the number of used bins.

Bonus question Compare different search strategies (variables/values selector) on this model for n between 10 and 15.

Take a look at the following exercise in the old version of Choco and try to transpose it on new version of Choco.

Here is **the complete code in Choco1** : BinPackingv1.zip.

```
int[] instance = getRandomPackingPb(n, capaBin, seed);
QuickSort sort = new QuickSort(instance); //Sort objects in increasing order
sort.sort();
Problem pb = new Problem();
IntDomainVar[] debut = new IntDomainVar[n];
IntDomainVar[] duree = new IntDomainVar[n];
IntDomainVar[] fin = new IntDomainVar[n];

int nbBinMin = computeLB(instance, capaBin);
for (int i = 0; i < n; i++) {
    debut[i] = pb.makeEnumIntVar("debut " + i, 0, n);
    duree[i] = pb.makeEnumIntVar("duree " + i, 1, 1);
    fin[i] = pb.makeEnumIntVar("fin " + i, 0, n);
}
IntDomainVar obj = pb.makeEnumIntVar("nbBin ", nbBinMin, n);
pb.post(pb.cumulative(debut, fin, duree, instance, capaBin));
for (int i = 0; i < n; i++) {
    pb.post(pb.geq(obj, debut[i]));
}

IntDomainVar[] branchvars = new IntDomainVar[n + 1];
System.arraycopy(debut, 0, branchvars, 0, n);
branchvars[n] = obj;

//long tps = System.currentTimeMillis();
pb.getSolver().setVarSelector(new StaticVarOrder(branchvars));
Solver.setVerbosity(Solver.SOLUTION);
pb.minimize(obj, false);
Solver.flushLogs();
// print solution
System.out.println("------------------------ " + (obj.getVal() + 1) + " bins");
if (pb.isFeasible() == Boolean.TRUE) {
    for (int j = 0; j <= obj.getVal(); j++) {
    System.out.print("Bin " + j + ": ");
    int load = 0;
    for (int i = 0; i < n; i++) {
        if (debut[i].isInstantiatedTo(j)) {
```

```
            System.out.print(i + "␣");
            load += instance[i];
            }
        }
        System.out.println("␣-␣load␣" + load);
        }
        //System.out.println("tps " + tps + " node "
         // + ((NodeLimit) pb.getSolver().getSearchSolver().limits.get(1)).getNbTot());
    }
```

(Solution)

12.2.2 Exercise 2.2 (Social golfer)

A group of golfers play once a week and are splitted into k groups of size s (there are therefore ks golfers in the club). The objective is to build a game scheduling on w weeks so that no golfer play in the same group than another one more than once (hence the name of the problem: *social golfers*). However, it may happen that two golfers will never play together. The point is only that once they have played together, they cannot play together anymore.

> You can test your model with the parameters (w, s, g) set to: $\{(11, 6, 2), (13, 7, 2), (9, 8, 8), (9, 8, 4), (4, 7, 3), (3, 6, 4)\}$.

Question 1 Propose a boolean model for this problem. Use an heuristic that consists in scheduling a golfer on every week before scheduling a new one. More precisely, a golfer can be put in the first available group of each week before considering the next golfer.

Question 2 Identify some symmetries of the problem by using every similar elements of the problem. Try to improve your model by breaking those symmetries.

	group 1	group 2	group 3	group 4
week 1	1 2 3	4 5 6	7 8 9	10 11 12
week 1	1 4 7	10 2 5	8 11 3	6 9 12
week 1	1 5 9	10 2 6	7 11 3	4 8 12

Table 12.1: A valid configuration with 4 groups of 3 golfers on 3 weeks.

Here is **the complete code in Choco1** : SocialGolferv1.zip

```
Problem pb = new Problem();
int numplayers = g * s;

// golfmat[i][j][k] : is golfer k playing week j in group i ?
IntDomainVar[][][] golfmat = new IntDomainVar[g][w][numplayers];
for (int i = 0; i < g; i++) {
    for (int j = 0; j < w; j++)
        for (int k = 0; k < numplayers; k++)
            golfmat[i][j][k] = pb.makeEnumIntVar("("+i+"_"+j+"_"+k+")", 0, 1);
}

//every week, every golfer plays in one group
for (int i = 0; i < w; i++) {
    for (int j = 0; j < numplayers; j++) {
        IntDomainVar[] vars = new IntDomainVar[g];
        for (int k = 0; k < g; k++) {
            vars[k] = golfmat[k][i][j];
        }
```

```
            pb.post(pb.eq(pb.scalar(vars, getOneMatrix(g)), 1));
        }
    }

    //every group is of size s
    for (int i = 0; i < w; i++) {
        for (int j = 0; j < g; j++) {
            IntDomainVar[] vars = new IntDomainVar[numplayers];
            System.arraycopy(golfmat[j][i], 0, vars, 0, numplayers);
            pb.post(pb.eq(pb.scalar(vars, getOneMatrix(numplayers)), s));
        }
    }

    //every pair of players only meets once
    // Efficient way : use of a ScalarAtMost
    for (int i = 0; i < numplayers; i++) {
        for (int j = i + 1; j < numplayers; j++) {
            IntDomainVar[] vars = new IntDomainVar[w * g * 2];
            int cpt = 0;
            for (int k = 0; k < w; k++) {
                for (int l = 0; l < g; l++) {
                    vars[cpt] = golfmat[l][k][i];
                    vars[cpt + w * g] = golfmat[l][k][j];
                    cpt++;
                }
            }
            pb.post(new ScalarAtMostv1(vars, w * g, 1));
        }
    }

    //break symetries among weeks
    //enforce a lexicographic ordering between every pairs of week
    for (int i = 0; i < w; i++) {
        for (int j = i + 1; j < w; j++) {
            IntDomainVar[] vars1 = new IntDomainVar[numplayers * g];
            IntDomainVar[] vars2 = new IntDomainVar[numplayers * g];
            int cpt = 0;
            for (int k = 0; k < numplayers; k++) {
                for (int l = 0; l < g; l++) {
                    vars1[cpt] = golfmat[l][i][k];
                    vars2[cpt] = golfmat[l][j][k];
                    cpt++;
                }
            }
            pb.post(pb.lex(vars1, vars2));
        }
    }

    //break symetries among groups
    for (int i = 0; i < numplayers; i++) {
        for (int j = i + 1; j < numplayers; j++) {
            IntDomainVar[] vars1 = new IntDomainVar[w * g];
            IntDomainVar[] vars2 = new IntDomainVar[w * g];
            int cpt = 0;
            for (int k = 0; k < w; k++) {
                for (int l = 0; l < g; l++) {
                    vars1[cpt] = golfmat[l][k][i];
                    vars2[cpt] = golfmat[l][k][j];
                    cpt++;
                }
            }
```

```
        pb.post(pb.lex(vars1, vars2));
    }
}

//break symetries among players
for (int i = 0; i < w; i++) {
    for (int j = 0; j < g; j++) {
        for (int p = j + 1; p < g; p++) {
            IntDomainVar[] vars1 = new IntDomainVar[numplayers];
            IntDomainVar[] vars2 = new IntDomainVar[numplayers];
            int cpt = 0;
            for (int k = 0; k < numplayers; k++) {
                vars1[cpt] = golfmat[j][i][k];
                vars2[cpt] = golfmat[p][i][k];
                cpt++;
            }
            pb.post(pb.lex(vars1, vars2));
        }
    }
}

//gather branching variables
IntDomainVar[] staticvars = new IntDomainVar[g * w * numplayers];
int cpt = 0;
for (int i = 0; i < numplayers; i++) {
    for (int j = 0; j < w; j++) {
        for (int k = 0; k < g; k++) {
            staticvars[cpt] = golfmat[k][j][i];
            cpt++;
        }
    }
}
pb.getSolver().setVarSelector(new StaticVarOrder(staticvars));

pb.getSolver().setTimeLimit(120000);
Solver.setVerbosity(Solver.SOLUTION);
pb.solve();
Solver.flushLogs();
```

(Solution)

12.2.3 Exercise 2.3 (Golomb rule)

under development

(Solution)

12.3 I know CP and Choco

12.3.1 Exercise 3.1 (Hamiltonian Cycle Problem Traveling Salesman Problem)

Given a graph $G = (V, E)$, an *Hamiltonian cycle* is a cycle that goes through every nodes of G once and only once. This exercise first intorduces a naive model to solve the Hamiltonian Cycle Problem. A second part tackles with the well known Traveling Salesman Problem.

Let $V = \{2, ..., n\}$ be a set of cities index to cover, and let d be a single warehouse duplicated into two indices 1 and $n+1$. Notice the duplication distinguishes the source from the sink while there is only one warehouse. Finally, let us denote by $V_d = V \cup \{1, n+1\}$ the set of nodes to cover by a tour. Thus, the two following problems are defined:

- find an Hamiltonian path covering all the cities of V

- find an Hamiltonian cycle of minimum cost that covers all the cities of V.

Question 1 [Hamiltonian Cycle Problem]:

We first consider the satisfaction problem. Formally, a directed graph $G = (V, E)$ represents the topology of the cities and the unfolded warehouse. There is an arc $(i, j) \in E$ iff there exists a directed road from $i \in V$ to $j \in V$. Furthermore, every arc $(1, i)$ and $(i, n + 1)$, with $i \in V$, belongs to E. Such a problem has to respect the following constraints:

- each node of V_d is reached exactly once,
- there is no subcycle containing nodes of V. In other words, the single cycle involved in G is hamiltonian and contains arc $(n + 1, 1)$.

Question 1.a The first constraint can directly be modelled using those proposed by Choco. On the other way, the second one requires to implement a constraint. This can be done through the following steps (see the provided skeleton):

- strictly specify your constraint signature,
- formalise the underlying subproblemand information that need to be maintained,
- ignore in a first time the Choco event based mechanism and implement your filtering algorithm directly within the `propagate()` method,
- once your algorithm has been checked, try to reformulate your constraint through an event based implementation with the following methods: `awakeOnInst()`, `awakeOnSup()`, `awakeOnInf()`, `awakeOnBounds()`, `awakeOnRem()`, `awakeOnRemovals()`.

Question 1.b Now, propose a search heuristic (both on variables and values) that incrementaly builds the searched path from the source node. For this purpose, you have to respectively implement java classes that inherit from `IntVarSelector` and `ValSelector`.

Question 2 [Traveling Salesman Problem]:

We now consider the optimisation view of the Hamiltonian Cycle Problem. A quantitative information is now associated with each arc of G given by a cost function $f : E \leftarrow \mathbb{Z}_+$. Then, the graph G is now defined by the triplet (V_d, E, f) and we have to find an Hamiltonian path of minimum cost in G.

For this purpose, we provide a skeleton of a Choco global constraint that dynamically maintains a lower bound evaluation of the searched path cost. Here, an evaluation of a minimum spanning tree of G is proposed. *Be careful*: take into account the partial assignment of the variables associated with the cities.

Question 2.a find an upper bound on the cost of the Hamiltonian path,

Question 2.b back-propagate lower/upper bounds informations on the required/infeasible arcs of G.

(Solution)

12.3.2 Exercise 3.2 (Shop scheduling)

Given a set of n tasks T and m disjunctive resources R, the problem is to find a plan to assign tasks to resources so that for every instant t, each resource $r \in R$ executes at most one task. Each task $T_i \in T$ is defined by:

- a starting date $s_i = [s_i^-, s_i^+] \in \mathbb{Z}_+$,
- an ending date $e_i = [e_i^-, e_i^+] \in \mathbb{Z}_+$,
- a duration $d_i = e_i - s_i$,
- a resource $r_i = \{res_1, \ldots, res_m\} \subseteq R$,

- a set of tasks, $preds_i \subseteq T$ that need to be processed before the start of T_i.

Let us consider the following satisfaction problem : Can one find a schedule of tasks T on the resources R that

- satisfies all the precedence constraints:

$$e_j \leq s_i, \qquad (\forall T_i \in T, \forall T_j \in preds_i)$$

- the last processed task ends before a given date D ?:

$$e_i \leq D, \qquad (\forall T_i \in T)$$

Then consider the optimization version : We now aim at finding the scheduling that satisfies all the constraints and that minimizes the date of the last task processed. You will be given for this :

- a class structure where you have to describe your model (`AssignmentProblem`),
- a class structure describing a Task (`Task`),
- a class structure (`BinaryNonOverlapping`) which defines a Choco constraint. This constraint takes two tasks as parameters and has to verify whether at any time t those tasks will be processed by the same resource or not.
- a class structure (`MandatoryInterval`) which describes for a given task, the time window it has to be processed in.

Question 1 How would you model the job scheduling problem ? Make use of the constraint `BinaryNonOverlapping`.

Question 2 Implement your model as if the constraint `BinaryNonOverlapping` was implemented.

Question 3 Sketch the mandatory processing interval of a task.

Question 4 Implement the constraint `BinaryNonOverlapping`:

- Implement the following reasoning : if two tasks have to be processed on the same resource and their mandatory intervals intersect, throw a failure.
- Now, implement the condition : if two tasks have a mandatory interval intersection, they must be scheduled on different resources.
- Finally, implement the following reasoning : If two tasks have to be processed by the same resource, then the starting and ending dates of every task ought to be updated functions to their mandatory intervals.

Question 5 Implement an variable selection heuristic on the decision variable of the problem.

Question 6 Propose a model which minimize the end date of the last assigned task.

Question 7 Can you find a way to improve the BinaryNonOverlapping constraint.

Bonus Question Find a lower bound on the end date of the last processed task.

Solution

Chapter 13

Solutions

13.1 I'm new to CP

13.1.1 Solution of Exercise 1.1 (A soft start)

(Problem)

***Question 1**: describe the constraint network related to code*

The model is defined as :

- $V = \{x_1, x_2, x_3\}$: the set of variables,
- $D = \{[0,5], [0,5], [0,5]\}$: the set of domain
- $C = \{x_1 > x_2, x_1 \neq x_3, x_2 > x_3\}$: the set of constraints.

***Question 2**: give the variable domains after constraint propagation.*

- From $x_1 = [0,5]$ and $x_2 = [0,5]$ and $x_1 > x_2$, we can deduce tha : the domain of x_1 can be reduce to $[1,5]$ and the domain of x_2 can be reduce to $[0,4]$.
- Then, from $x_2 = [0,4]$ and $x_3 = [0,5]$ and $x_2 > x_3$, we can deduce that : the domain of x_2 can be reduce to $[1,4]$ and the domain of x_3 can be reduce to $[0,3]$.
- Then, from $x_1 = [1,5]$ and $x_2 = [1,4]$ and $x_1 > x_2$, we can deduce that : the domain of x_1 can be reduce to $[2,5]$.

We cannot deduce anything else, so we have reached a **fix point**, and here is the domain of each variables:

$$x_1 : [2,5], \quad x_2 : [1,4], \quad x_3 : [0,3].$$

13.1.2 Solution of Exercise 1.2 (DONALD + GERALD = ROBERT)

(Problem)

Source code: ExDonaldGeraldRobert.zip

```
// Build model
Model model = new CPModel();

// Declare every letter as a variable
IntegerVariable d = makeIntVar("d", 0, 9, "cp:enum");
IntegerVariable o = makeIntVar("o", 0, 9, "cp:enum");
IntegerVariable n = makeIntVar("n", 0, 9, "cp:enum");
IntegerVariable a = makeIntVar("a", 0, 9, "cp:enum");
IntegerVariable l = makeIntVar("l", 0, 9, "cp:enum");
IntegerVariable g = makeIntVar("g", 0, 9, "cp:enum");
IntegerVariable e = makeIntVar("e", 0, 9, "cp:enum");
```

```
IntegerVariable r = makeIntVar("r", 0, 9, "cp:enum");
IntegerVariable b = makeIntVar("b", 0, 9, "cp:enum");
IntegerVariable t = makeIntVar("t", 0, 9, "cp:enum");

// Declare every name as a variable
IntegerVariable donald = makeIntVar("donald", 0, 1000000,"cp:bound");
IntegerVariable gerald = makeIntVar("gerald", 0, 1000000,"cp:bound");
IntegerVariable robert = makeIntVar("robert", 0, 1000000,"cp:bound");

// Array of coefficients
int[] c = new int[]{100000, 10000, 1000, 100, 10, 1};

// Declare every combination of letter as an integer expression
IntegerExpressionVariable donaldLetters = scalar(new IntegerVariable[]{d,o,n,a,l,d}, c);
IntegerExpressionVariable geraldLetters = scalar(new IntegerVariable[]{g,e,r,a,l,d}, c);
IntegerExpressionVariable robertLetters = scalar(new IntegerVariable[]{r,o,b,e,r,t}, c);

// Add equality between name and letters combination
model.addConstraint(eq(donaldLetters, donald));
model.addConstraint(eq(geraldLetters, gerald));
model.addConstraint(eq(robertLetters, robert));
// Add constraint name sum
model.addConstraint(eq(plus(donald, gerald), robert));
// Add constraint of all different letters.
model.addConstraint(allDifferent(new IntegerVariable[]{d,o,n,a,l,g,e,r,b,t}));

// Build a solver, read the model and solve it
Solver s = new CPSolver();
s.read(model);
s.solve();

// Print name value
System.out.println("donald = " + s.getVar(donald).getVal());
System.out.println("gerald = " + s.getVar(gerald).getVal());
System.out.println("robert = " + s.getVar(robert).getVal());
```

13.1.3 Solution of Exercise 1.3 (A famous example. . . a sudoku grid)

(Problem)

Source code: ExSudoku.zip

Question 1: *propose a way to model the sudoku problem with difference constraints. Implement your model with choco solver.*

```
int n = instance.length;
// Build Model
Model m = new CPModel();

// Build an array of integer variables
IntegerVariable[][] rows = makeIntVarArray("rows", n, n, 1, n,"cp:enum");

// Not equal constraint between each case of a row
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        for (int k = j; k < n; k++)
            if (k != j) m.addConstraint(neq(rows[i][j], rows[i][k]));
}

// Not equal constraint between each case of a column
for (int j = 0; j < n; j++) {
    for (int i = 0; i < n; i++)
```

```
        for (int k = 0; k < n; k++)
            if (k != i) m.addConstraint(neq(rows[i][j], rows[k][j]));
}

// Not equal constraint between each case of a sub region
for (int ci = 0; ci < n; ci += 3) {
    for (int cj = 0; cj < n; cj += 3)
        // Extraction of disequality of a sub region
        for (int i = ci; i < ci + 3; i++)
            for (int j = cj; j < cj + 3; j++)
                for (int k = ci; k < ci + 3; k++)
                    for (int l = cj; l < cj + 3; l++)
                        if (k != i || l != j) m.addConstraint(neq(rows[i][j], rows[k][l]));
}

//...

// Call solver
Solver s = new CPSolver();
s.read(m);
CPSolver.setVerbosity(CPSolver.SOLUTION);
s.solve();
CPSolver.flushLogs();
printGrid(rows, s);
```

***Question 2**: which global constraint can be used to model such a problem ? Modify your code to use this constraint.*

The *allDifferent* constraint can be used to remplace every disequality constraint on the first Sudoku model. It improves the efficient of the model and make it more "readable".

```
// Build model
Model m = new CPModel();
// Declare variables
IntegerVariable[][] cols = new IntegerVariable[n][n];
IntegerVariable[][] rows = makeIntVarArray("rows", n, n, 1, n,"cp:enum");

// Channeling between rows and columns
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        cols[i][j] = rows[j][i];
}

// Add alldifferent constraint
for (int i = 0; i < n; i++) {
    m.addConstraint(allDifferent(cols[i]));
    m.addConstraint(allDifferent(rows[i]));
}

// Define sub regions
IntegerVariable[][] carres = new IntegerVariable[n][n];
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++)
        for (int k = 0; k < 3; k++)
            carres[j + k * 3][i] = rows[0 + k * 3][i + j * 3];
            carres[j + k * 3][i + 3] = rows[1 + k * 3][i + j * 3];
            carres[j + k * 3][i + 6] = rows[2 + k * 3][i + j * 3];
}

// Add alldifferent on sub regions
for (int i = 0; i < n; i++) {
    Constraint c = allDifferent(carres[i]);
```

```
    m.addConstraint(c);
}

//...

// Call solver
Solver s = new CPSolver();
s.read(m);
CPSolver.setVerbosity(CPSolver.SOLUTION);
s.solve();
printGrid(rows, s);
```

***Question 3**: Test for both model the initial propagation step (use choco* `propagate()` *method). What can be noticed ? What is the point in using global constraints ?*

The sudoku problem can be solved just with the propagation. FIXME explanation. The global constraint provides a more efficient filter algorithm, due to more complex deduction.

13.1.4 Solution of Exercise 1.4 (The knapsack problem)

(Problem)

Source code: ExKnapSack.zip

***Question 1** : In the first place, we will not consider the idea of maximizing the energetic value. Try to find a satisfying solution by modelling and implementing the problem within choco.*

```
Model m = new CPModel();

obj1 = makeIntVar("obj1", 0, 5,"cp:enum");
obj2 = makeIntVar("obj2", 0, 7,"cp:enum");
obj3 = makeIntVar("obj3", 0, 10,"cp:enum");
c = makeIntVar("cost", 1, 1000000,"cp:bound");

int capacity = 34;
int[] volumes = new int[]{7, 5, 3};
int[] energy = new int[]{6, 4, 2};

m.addConstraint(leq(scalar(volumes, new IntegerVariable[]{obj1, obj2, obj3}), capacity));
m.addConstraint(eq(scalar(energy, new IntegerVariable[]{obj1, obj2, obj3}), c));

Solver s = new CPSolver();
s.read(m);

s.solve();

System.out.println("("+s.getVar(obj1).getVal()+","+s.getVar(obj2).getVal()+","
                   +s.getVar(obj3).getVal()+")␣cost␣=␣"+ s.getVar(c).getVal());
```

***Question 2** : Find and use the choco method to maximise the energetic value of the knapsack.* Replace `s.solve()` by:

```
s.maximize(s.getVar(c), false);
```

***Question 3** : Propose a Value selector heuristic to improve the efficiency of the model.*

It can be improved using the following value selector strategy. It iterates over decreasing values of every domain variables:

```
s.setValIntIterator(new DecreasingDomain());
```

13.1.5 Solution of Exercise 1.5 (The n-queens problem)

(Problem)

Source code: ExQueen.zip

Question 1 *: propose and implement a model based on one L_i variable for every row...*

```
Model m = new CPModel();

IntegerVariable[] queens = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queens[i] = makeIntVar("Q" + i, 1, n,"cp:enum");
}

for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        int k = j - i;
        m.addConstraint(neq(queens[i], queens[j]));
        m.addConstraint(neq(queens[i], plus(queens[j], k))); // diagonal
        m.addConstraint(neq(queens[i], minus(queens[j], k))); // diagonal
    }
}

Solver s = new CPSolver();
s.read(m);
CPSolver.setVerbosity(CPSolver.SOLUTION);
int timeLimit = 60000;
s.setTimeLimit(timeLimit);
s.solve();
CPSolver.flushLogs();
```

Question 2 *: Add a redundant model by considering variable on the columns (C_i). Continue to use simple difference constraints.*

```
Model m = new CPModel();

IntegerVariable[] queens = new IntegerVariable[n];
IntegerVariable[] queensdual = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queens[i] = makeIntVar("Q" + i, 1, n,"cp:enum");
    queensdual[i] = makeIntVar("QD" + i, 1, n,"cp:enum");
}

for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        int k = j - i;
        m.addConstraint(neq(queens[i], queens[j]));
        m.addConstraint(neq(queens[i], plus(queens[j], k))); // diagonal
        m.addConstraint(neq(queens[i], minus(queens[j], k))); // diagonal
    }
}

for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        int k = j - i;
        m.addConstraint(neq(queensdual[i], queensdual[j]));
        m.addConstraint(neq(queensdual[i], plus(queensdual[j], k))); // diagonal
        m.addConstraint(neq(queensdual[i], minus(queensdual[j], k))); // diagonal
    }
}
m.addConstraint(inverseChanneling(queens, queensdual));

Solver s = new CPSolver();
s.read(m);

s.setVarIntSelector(new MinDomain(s,s.getVar(queens)));
```

```
CPSolver.setVerbosity(CPSolver.SOLUTION);
s.setLoggingMaxDepth(50);
int timeLimit = 60000;
s.setTimeLimit(timeLimit);
s.solve();
CPSolver.flushLogs();
```

Question 3 *: Compare the number of nodes created to find the solutions with both models. How can you explain such a difference ?*

The channeling permit to reduce more nodes from the tree search... FIXME

Question 4 *: Add to the previous implemented model the following heuristics,*

- *Select first the line variable (L_i) which has the smallest domain ;*
- *Select the value $j \in L_i$ so that the associated column variable C_j has the smallest domain.*

Again, compare both approaches in term of nodes number and solving time to find ONE solution for $n = 75, 90, 95, 105$.

Add the following lines to your program (after the reading of the model):

```
s.setVarIntSelector(new MinDomain(s,s.getVar(queens)));
s.setValIntSelector(new NQueenValueSelector(s.getVar(queensdual)));
```

The variable selector strategy (`MinDomain`) already exists in Choco. It iterates over variables given and returns the variable ordering by creasing domain size. The value selector strategy has to be created as follow:

```
public class NQueenValueSelector implements ValSelector {

    // Column variable
    protected IntDomainVar[] dualVar;

    // Constructor of the value selector,
    public NQueenValueSelector(IntDomainVar[] cols) {
        this.dualVar = cols;
    }

    // Returns the "best val" that is the smallest column domain size OR -1
    // (-1 is not in the domain of the variables)
    public int getBestVal(IntDomainVar intDomainVar) {
        int minValue = 10000;
        int v0 = -1;
        IntIterator it = intDomainVar.getDomain().getIterator();
        while (it.hasNext()){
            int i = it.next();
            int val = dualVar[i - 1].getDomainSize();
            if (val < minValue) {
                minValue = val;
                v0 = i;
            }
        }
        return v0;
    }
}
```

Question 5 *: what changes are caused by the use of the global constraint* ***alldifferent*** *?*

```
Model m = new CPModel();

IntegerVariable[] queens = new IntegerVariable[n];
IntegerVariable[] queensdual = new IntegerVariable[n];
IntegerVariable[] diag1 = new IntegerVariable[n];
```

```
IntegerVariable[] diag2 = new IntegerVariable[n];
IntegerVariable[] diag1dual = new IntegerVariable[n];
IntegerVariable[] diag2dual = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queens[i] = makeIntVar("Q" + i, 1, n,"cp:enum");
    queensdual[i] = makeIntVar("QD" + i, 1, n,"cp:enum");
    diag1[i] = makeIntVar("D1" + i, 1, 2 * n,"cp:enum");
    diag2[i] = makeIntVar("D2" + i, -n, n,"cp:enum");
    diag1dual[i] = makeIntVar("D1" + i, 1, 2 * n,"cp:enum");
    diag2dual[i] = makeIntVar("D2" + i, -n, n,"cp:enum");
}

m.addConstraint(allDifferent(queens));
m.addConstraint(allDifferent(queensdual));
for (int i = 0; i < n; i++) {
    m.addConstraint(eq(diag1[i], plus(queens[i], i)));
    m.addConstraint(eq(diag2[i], minus(queens[i], i)));
    m.addConstraint(eq(diag1dual[i], plus(queensdual[i], i)));
    m.addConstraint(eq(diag2dual[i], minus(queensdual[i], i)));
}
m.addConstraint(inverseChanneling(queens,queensdual));

m.addConstraint(allDifferent(diag1));
m.addConstraint(allDifferent(diag2));
m.addConstraint(allDifferent(diag1dual));
m.addConstraint(allDifferent(diag2dual));

Solver s = new CPSolver();
s.read(m);

s.setVarIntSelector(new MinDomain(s,s.getVar(queens)));
s.setValIntSelector(new NQueenValueSelector(s.getVar(queensdual)));

CPSolver.setVerbosity(CPSolver.SOLUTION);
int timeLimit = 60000;
s.setTimeLimit(timeLimit);
s.solve();
CPSolver.flushLogs();
```

13.2 I know CP

13.2.1 Solution of Exercise 2.1 (Bin packing, cumulative and search strategies)

(Problem)

Source code: BinPackingv2.zip

13.2.2 Solution of Exercise 2.2 (Social golfer)

(Problem)

Source code: SocialGolferv2.zip

13.2.3 Solution of Exercise 2.3 (Golomb rule)

under development

(Problem)

13.3 I know CP and Choco2.0

13.3.1 Solution of Exercise 3.1 (Hamiltonian Cycle Problem Traveling Salesman Problem)

(Problem)
Source code: ExTSP.zip

13.3.2 Solution of Exercise 3.2 (Shop scheduling)

(Problem)
under development

Part IV

Extras

Chapter 14

Choco and Visu

14.1 Why?

Since few months, it has seemed more and more evident for us that CHOCO needed a way to visualize dynamically the resolution of a problem. We wanted that visualization to be open, easy to use and not static. Now, you will find a new package on **Choco 2.0.1** (the actual beta version) named *visu*.

14.2 The visu package

The *visu* package contains objects to define a visualization of the resolution, domain reduction, constraints propagation, etc.

Figures **??** depicts the class diagram of the visu package (*powered by BOUML*):

14.3 Steps to use the Visu

Only one Visu can be linked to one Solver.

We are going to see a short example of Visu use, based on Sudoku problem. In our modeling, variables are cells of a sudoku grid, represented by the matrix *rows*. We want to define a standard visualization where a variable is displayed on a line. Its name is written, and the domain is viewed as an array of colored square. That representation is known in CHOCO as a *FULLDOMAIN* representation.

14.3.1 Visu creation

The first step is to create the Visu object, which is basically a frame with components. We use the static constructor defined in `Visu.java`:

- `Visu.createFullVisu()`: build a Visu object with default minimum size (width 480 px and heigth 640 px), with *next*, *play*,*pause* buttons and the break length slider.
- `Visu.createFullVisu(int width, int height)`: build a Visu object with user defined minimum size (width *width* px and heigth *height* px), with *next*, *play*,*pause* buttons and the break length slider.
- `Visu.createVisu(VisuButton... buttons)`: build a Visu object with default minimum size (width 480 px and heigth 640 px), with *buttons* buttons and the break length slider.
- `Visu.createVisu(int width, int height, final VisuButton... buttons)`: build a Visu object with user defined minimum size (width *width* px and heigth *height* px), with *buttons* buttons and the break length slider if necessary (at least, if there.is one button).

Parameter *buttons* is an array of VisuButton that can take one of the following values: *NEXT*, *PLAY*. *NEXT* add the *next* button to the frame and the slider, *PLAY* add the *play* and *pause* buttons and the slider.

We want to create a simple full Visu:

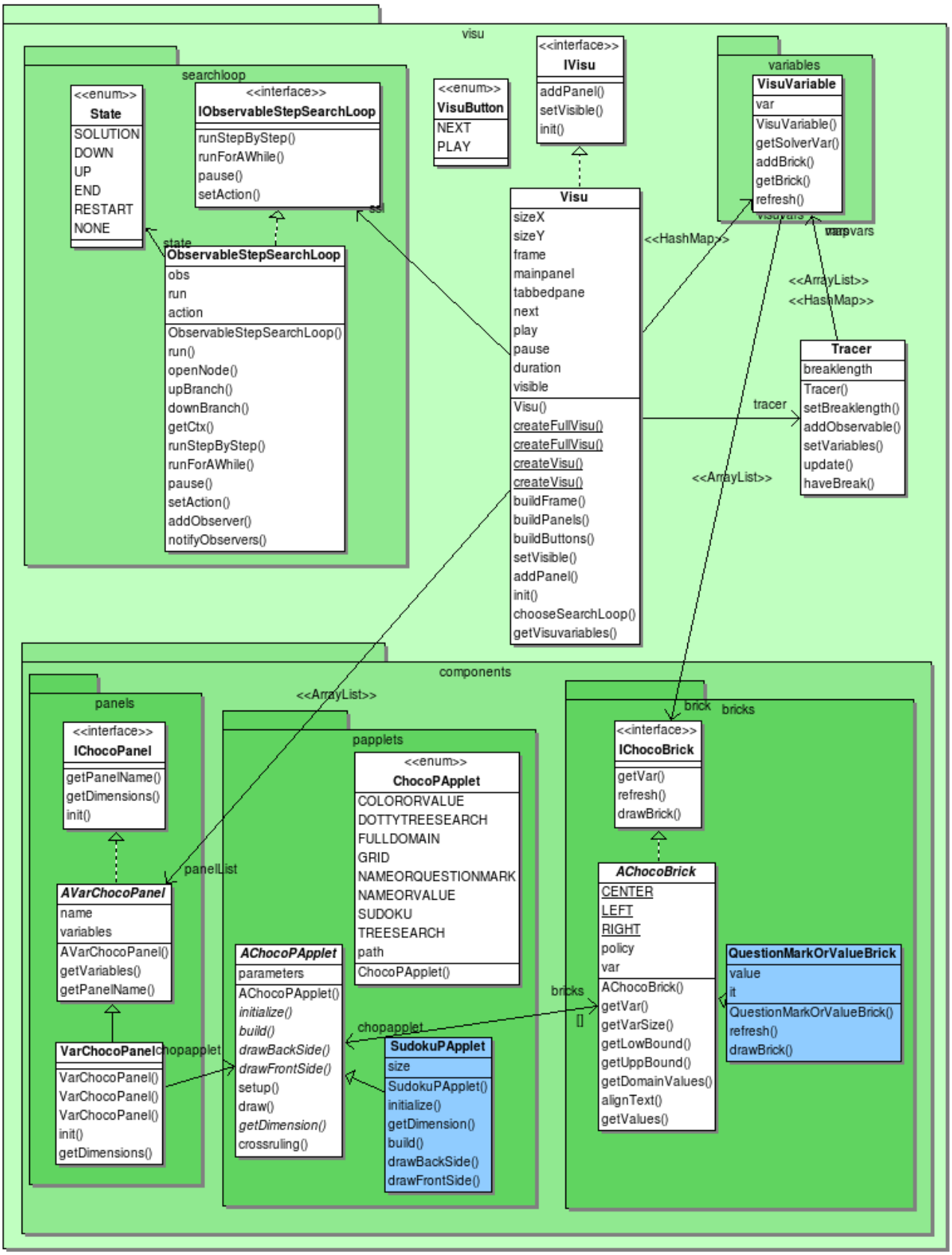

Figure 14.1: Visu classes diagram. The blue classes are examples of implementation and inheritence.

```
Visu v = Visu.createVisu();
```

14.3.2 Adding panel

Now the frame is defined, we have to add a component: a `VarChocoPanel`. It is a specified panel, added to a `TabbedPane`, where one visualization (a `ChocoPApplet`) can be put. A `ChocoPApplet` can be defined in two ways: an existing one, or a user defined one. Constructors of `VarChocoPanel` are:

- `VarChocoPanel(final String name, final Variable[] x, final ChocoPApplet applet, final Object params)`: to add a predefined `ChocoPApplet`. *params* can be null, except for *applet=DOTTYTREESEARCH* (see below).
- `VarChocoPanel(final String name, final Variable[] x, final Class appletclass, Object params)`: like previous, but `ChocoPApplet` is replaced by *class* which is the class name of the user's `ChocoPApplet`. Recommanded for use of user's ChocoPApplet.
- `VarChocoPanel(final String name, final Variable[] x, final String appletpath, Object params)`: like previous, but `ChocoPApplet` is replaced by *path* which is the path of the user's `ChocoPApplet` in the project.

Existing ChocoPApplet

Few ChocoPApplet are defined in Choco:

- **COLORORVALUE** : draw an applet where variables are in columns and where their value is displayed with a colored square (blue: not instantiated, green: instantiated),
- **DOTTYTREESEARCH** : specific applet, which do not display anything, but a *screensaver*. It builds a dot file (name given in parameters) with nodes of the tree search, to represent the tree search. The paramaters are :
  - *filename* (`String`) : output file name
  - *nbMaxNode* (`int`): size limit of the tree seach. If there is more than *nbMaxNode* nodes, the dot file will not be printed. The number of nodes has an impact on the file size
  - *watch* (`Var`) : the variable to optimize. Can be `null` if no optimization is performed.
  - *maximize* (`Boolean`) : indicating wether the optimization is a maximization (if set to `true`) or a minimization (if set to `false`). Can be `null` if no optimization is performed.
  - *restart* (`Boolean`) : indicating wether the search can restart (is set to `true`) or not (if set to `false`). Can be `null` if no optimization is performed.
- **FULLDOMAIN** : draw an applet where variables are in columns. Each line is build with a variable name and a set of colored square (blue: not instantiated, green: instantiated) representing each value of the domain.
- **GRID** : draw an applet with a simple grid, where each cells contains the value of a variable (question mark or value).
- **NAMEORQUESTIONMARK** : draw an applet where a variables are displayed on columns, by a question mark (if not instanciated) or its value (if instanciated).
- **NAMEORVALUE** : draw an applet where a variables are displayed on columns, by its name (if not instanciated) or its value (if instanciated).
- **SUDOKU** : specific applet, draw a sudoku grid where each cell represents the value of a variable or a question mark.
- **TREESEARH** : draw the dynamique construction of the tree search.

To add a panel where one of that ChocoPApplet will be drawn, use the following code:

```
Visu v = Visu.createVisu(
v.addPanel(new VarChocoPanel("Grid", vars, GRID, null));
v.addPanel(new VarChocoPanel("TreeSearch", vars, TREESEARCH, null));
v.addPanel(new VarChocoPanel("Dotty", vars, DOTTYTREESEARCH,
          new Object[]{"/home/choco/treesearch.dot", 100, null, null, null}));
```

User ChocoPApplet

UNDER DEVELOPMENT

14.4 Examples

UNDER DEVELOPMENT

Chapter 15

Sudoku and Constraint Programming

15.1 Sudoku ?!?

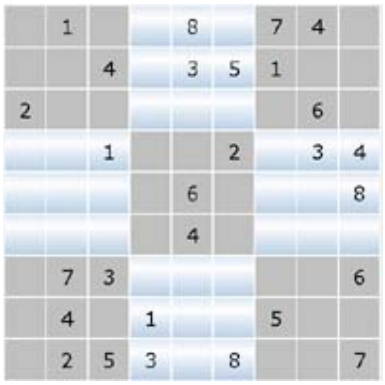

Figure 15.1: A sudoku grid

Everybody knows those grids that appeared last year in the subway, in wating lounges, on colleague's desks, etc. In Japanese *su* means digit and *doku*, unique. But this game has been discovered by an American ! The first grids appeared in the USA in 1979 (they were hand crafted). Wikipedia tells us that they were designed by Howard Garns a retired architect. He died in 1989 well before the success story of sudoku initiated by Wayne Gould, a retired judge from Hong-Kong. The rules are really simple: a 81 cells square grid is divided in 9 smaller blocks of 9 cells (3 x 3). Some of the 81 are filled with one digit. The aim of the puzzle is to fill in the other cells, using digits except 0, such as each digit appears once and only once in each row, each column and each smaller block. The solution is unique.

15.1.1 Solving sudokus

Many computer techniques exist to quickly solve a sudoku puzzle. Mainly, they are based on backtracking algorithms. The idea is the following: give a free cell a value and continue as long as choices remain consistent. As soon as an inconsistency is detected, the computer program backtracks to its earliest past choice et tries another value. If no more value is available, the program keeps backtracking until it can go forward again. This systematic technique make it sure to solve a sudoku grid. However, no human player plays this way: this needs too much memory !

> see Wikipedia for a panel of solving techniques.

15.2 Sudoku and Artificial Intelligence

Many techniques and rules have been designed and discovered to solve sudoku grids. Some are really simple, some need to use some useful tools: pencil and eraser.

15.2.1 Simple rules: single candidate and single position

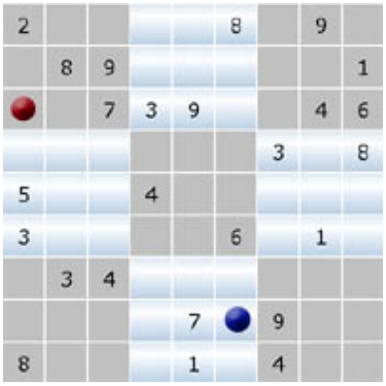

Figure 15.2: Simple rules: single candidates and single position

Let consider the grid on Figure **??** and the cell with the red dot. In the same line, we find: 3, 4, 6, 7, and 9. In the same column: 2, 3, 5, and 8. In the same block: 2, 7, 8, and 9. There remain only one possibility: **1**. This is the **single candidate** rule. This cell should be filled in with **1**.

Now let consider a given digit: let's say 4. In the block with a blue dot, there is no 4. Where can it be ? The 4's in the surrounding blocks heavily constrain the problem. There is a **single position** possible: the blue dot. This another simple rule to apply.

Alternatively using these two rules allows a player to fill in many cells and even solve the simplest grids. But, limits are easily reached. More subtle approaches are needed: but an important tool is now needed ... an eraser !

15.2.2 Human reasoning principles

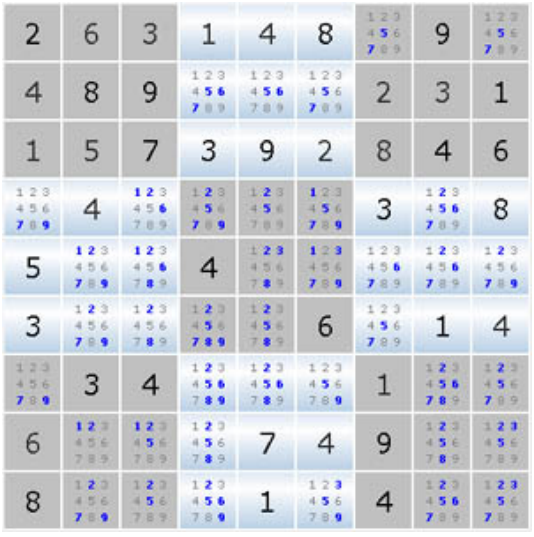

Figure 15.3: Introducing marks

Many techniques do exist but a vaste amount of them rely on simple principles. The first one is: do not try to find the value of a cell but instead focus on values that **will never be assigned** to it. The

space of possibility is then reduced. This is where the eraser comes handy. Many players marks the remaining possibilities as in the grid on the left.

Using this information, rather subtle reasoning is possible. For example, consider the seventh column on the grid on the left. Two cells contain as possible values the two values 5 and 7. This means that those two values cannot appear elsewhere in that very same column. Therefore, the other unassigned cell on the column can only contain a 6. We have *deduced* something.

This was an easy to spot inference. This is not always the case. Consider the part of the grid on the right. Let us consider the third column. For cells 4 and 5, only two values are available: 4 and 8. Those values cannot be assigned to any other cell in that column. Therefore, in cell 6 we have a 3, and thus and 7 in cell 2 and finally a 1 in cell 3. This can be a very powerful rule.
Such a reasoning (sometimes called *Naked Pairs*) is easily generalized to any number of cells (always in the same region: row, column or block) presenting this same configuration. This local reasoning can be applied to any region of the grid. It is important to notice that the inferred information can (and should) be used from a region to another.

The following principles of *human* reasoning can be listed:

- reasoning on *possible* values for a cell (by erasing impossible ones)
- systematically applying an evolved local reasoning (such as the *Naked Pairs* rule)
- transmitting inferred information from a region to another related through a given a set of cells

15.2.3 Towards Constraint Programming

Those three principles are at the core of **constraint programming** a recent technique coming from both *artificial intelligence* and *operations research*.

- The first principle is called **domain reduction** or *filtering*
- The second considers its region as a **constraint** (a relation to be verified by the solution of the problem): here we consider an *all different* constraint (all the values must be different in a given region). Constraints are considered **locally** for reasoning
- The third principle is called **propagation**: constraints (regions) communicate with one another through the available values in variables (cells)

Constraint programming is able to solve this problem as a human would do. Moreover, a large majority of the rules and techniques described on the Internet amount to a well-known problem: the alldifferent problem. A **constraint solver** (as **Choco**) is therefore able to reason on this problem allowing the solving of sudoku grid as a human would do although it has not be specifically designed to.

Ideally, iterating local reasoning will lead to a solution. However, for exceptionnaly hard grids, an enumerating phase (all constraint solvers provide tools for that) relying on backtracking may be necessary.

15.3 See also

- SudokuHelper a sudoku solver and helper applet developed with *Choco*.
- PalmSudoku a rather complete list of rules and tips for solving sudokus