Choco solver

http://choco.emn.fr/

Documentation

October 28, 2010

Contents

Preface

Choco is a java library for constraint satisfaction problems (CSP) and constraint programming (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 informations visit `http://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, and some advanced usages (customizing propagation and search).
- Elements of Choco gives a detailed description of the variables, operators, constraints currently available in Choco.
- 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, backtrackable structure and write logging statements.
-

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 $X^j = \{X^j_1, X^j_2, \ldots, X^j_{n^j}\} \subseteq X$ of variables which restricts the possible tuples of values $(v_1, \ldots, v_{n^j})$ for these variables:

  $$(v_1, \ldots, v_{n^j}) \in C_j \cap (D(X^j_1) \times D(X^j_2) \times \cdots \times D(X^j_{n^j})).$$

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

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

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

For optimization problems, one need to define an **objective function** $f : D(X) \to \mathbb{R}$. 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 $X_i \in X$, $D(X_i) = \{1, 2, \dots, n\}$.
- **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**:

$$\begin{aligned} y^2 * (1 + z^2) + z * (z - 24 * y) = -13 \\ x^2 * (1 + y^2) + y * (y - 24 * x) = -13 \\ z^2 * (1 + x^2) + x * (x - 24 * z) = -13 \end{aligned}$$

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:

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

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 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
    CPModel 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;
```

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(Choco.eq(Choco.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(Choco.eq(Choco.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(Choco.eq(Choco.sum(varDiag1), magicSum));
        m.addConstraint(Choco.eq(Choco.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
        CPSolver s = new CPSolver();
```

with the imports:

```
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(MessageFormat.format("{0}␣", 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.

```
int nbQueen = 8;
//1- Create the model
CPModel m = new CPModel();
//2- Create the variables
IntegerVariable[] queens = Choco.makeIntVarArray("Q", nbQueen, 1, nbQueen);
//3- Post constraints
for (int i = 0; i < nbQueen; i++) {
   for (int j = i + 1; j < nbQueen; j++) {
      int k = j - i;
      m.addConstraint(Choco.neq(queens[i], queens[j]));
      m.addConstraint(Choco.neq(queens[i], Choco.plus(queens[j], k))); // diagonal
          constraints
      m.addConstraint(Choco.neq(queens[i], Choco.minus(queens[j], k))); // diagonal
          constraints
   }
}
//4- Create the solver
CPSolver s = new CPSolver();
s.read(m);
s.solveAll();
//5- Print the number of solutions found
System.out.println("Number␣of␣solutions␣found:"+s.getSolutionCount());
```

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
CPModel 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] = Choco.makeSetVar("set␣" + i, 1, n);
for (int i = 0; i < n; i++)
   for (int j = i + 1; j < n; j++)
      intersect[i * n + j] = Choco.makeSetVar("interSet␣" + i + "␣" + j, 1, n);

//3- Post constraints
for (int i = 0; i < n; i++)
   mod.addConstraint(Choco.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(Choco.setInter(vars[i], vars[j], intersect[i * n + j]));
      mod.addConstraint(Choco.leqCard(intersect[i * n + j], 1));
   }

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

//5- Print the solution found
for(SetVariable var: vars){
        System.out.println(s.getVar(var).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.

```
    //1- Create the problem

    CPModel pb = new CPModel();

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

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

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

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

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

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

    //4- Search for all solution
    CPSolver s = new CPSolver();
    s.getConfiguration().putDouble(Configuration.REAL_PRECISION, 1e-8);
    s.read(pb);
    s.setVarRealSelector(new CyclicRealVarSelector(s));
    s.setValRealIterator(new RealIncreasingDomain());
    s.solve();
    //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(var1);
      model.addVariables(var2, var3);
```

Explictly addition of variables is not mandatory. See `Constraint` for more details.

Specific role of variables *var* can be defined with *options*: non-decision variables or objective variable;

```
      model.addVariable(Options.V_OBJECTIVE, var4);
      model.addVariables(Options.V_NO_DECISION, var5, var6);
```

2.1.1 Simple Variables

See Section Variables for details:

IntegerVariable, SetVariable, RealVariable

2.1.2 Constants

A constant is a variable with a fixed domain. An IntegerVariable declared with a unique value is automatically set as constant. A constant declared twice or more is only stored once in a model.

```
IntegerConstantVariable c10 = Choco.constant(10);
RealConstantVariable c0dot0 = Choco.constant(0.0);
SetConstantVariable c0_12 = Choco.constant(new int[]{0, 12});
SetConstantVariable cEmpty = Choco.emptySet();
```

2.1.3 Expression variables and operators

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

IntegerExpressionVariable and RealExpressionVariable.

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

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

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

abs, cos, distEq, distGt, distLt, distNeq, div, ifThenElse, max, min, minus, mod, mult, neg, plus, power, scalar, sin, sum.

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

2.1.4 MultipleVariables

These are syntaxic sugar. To make their declaration easier, tree, geost, and scheduling constraints allow or require to use multiple variables, like TreeParametersObject, GeostObject or TaskVariable. See also the code examples for these constraints.

2.1.5 Decision/non-decision variables

By default, each variable added to a model is a decision variable, *i.e.* is included in the default search strategy. A variable can be stated as a non decision one if its value can be computed by side-effect. To specify non decision variables, one can

- exclude them from its search strategies (see search strategy for more details);
- specify non-decision variables (adding Options.V_NO_DECISION to their options) and keep the default search strategy.

```
IntegerVariable vNoDec = Choco.makeIntVar("vNoDec", 1, 3, Options.V_NO_DECISION);
```

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

```
IntegerVariable vNoDec1 = Choco.makeIntVar("vNoDec1", 1, 3);
IntegerVariable vNoDec2 = Choco.makeIntVar("vNoDec2", 1, 3);
model.addOptions(Options.V_NO_DECISION, vNoDec1, vNoDec2);
```

> The declaration of a search strategy will erase setting `Options.V_NO_DECISION`.

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

2.1.6 Objective variable

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

```
IntegerVariable x = Choco.makeIntVar("x", 1, 1000, Options.V_OBJECTIVE);
IntegerVariable y = Choco.makeIntVar("y", 20, 50);
model.addConstraint(Choco.eq(x, Choco.mult(y, 20)));
solver.read(model);
solver.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.)

2.2 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:

```
model.addConstraint(c1);
model.addConstraints(c2,c3);
```

> Adding a constraint automatically adds its variables to the model (explicit declaration of variables addition is not mandatory).

Example:

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

```
model.addConstraint(Choco.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. See Section options ans settings for more details, specific APIs exist for declaring options constraints.

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 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.

- `member`, `notMember`
- `eqCard`, `geqCard`, `leqCard`, `neqCard`
- `eq`
- `isIncluded`, `isNotIncluded`
- `setInter`
- `setDisjoint`, `setUnion`
- `max`, `min`
- `inverseSet`
- `among`
- `pack`

2.2.5 Channeling constraints

The use of a redundant model, based on an alternative set of decision variables, is a frequent technique to strengthen propagation or to get more freedom to design dedicated search heuristics. The following constraints allow to ensure the integrity of two redundant models by linking (channeling) variable-value assignments in the first model to variable-value assignments in the second model:

- `boolChanneling` $b_j = 1 \iff x = j$,
- `domainChanneling` $b_j = 1 \iff x = j, \ \forall j$,
- `inverseChanneling` $y_j = i \iff x_i = j, \ \forall i, j$,
- `inverseSet` $i \in s_j \iff x_i = j, \ \forall i, j$,

In the n-queen problem, for example, a domain variable by column indicates the row j to place a queen in column i. To enhance the propagation, a redundant model can be stated by defining a domain variable by row indicating the column i. As columns and rows can be interchanged, the same set of constraints applies to both models, then constraint inverseChanneling is set to propagate between the two models.

```
int n = 8;
Model m = new CPModel();
IntegerVariable[] queenInCol = new IntegerVariable[n];
IntegerVariable[] queenInRow = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queenInCol[i] = makeIntVar("QC" + i, 1, n);
    queenInRow[i] = makeIntVar("QR" + 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(queenInCol[i], queenInCol[j])); // row
        m.addConstraint(neq(queenInCol[i], plus(queenInCol[j], k))); // diagonal 1
        m.addConstraint(neq(queenInCol[i], minus(queenInCol[j], k))); // diagonal 2
        m.addConstraint(neq(queenInRow[i], queenInRow[j])); // column
        m.addConstraint(neq(queenInRow[i], plus(queenInRow[j], k))); // diagonal 2
        m.addConstraint(neq(queenInRow[i], minus(queenInRow[j], k))); // diagonal 1
    }
}
m.addConstraint(inverseChanneling(queenInCol, queenInRow));
Solver s = new CPSolver();
s.read(m);
s.solveAll();
```

Channeling constraints are also useful to compose a model made of two parts as, for example, in a task-resources assignment problem where some constraints are set on the task set and some other constraints are set on the resource set.

Reification offers an other type of channeling, between a constraint and a boolean variable representing its truth value. More complex channeling can be done using reification and boolean operators although they are less effective. The reified constraint below states $b = 1 \iff x = y$:

```
 IntegerVariable b = Choco.makeBooleanVar("b");
IntegerVariable x = Choco.makeIntVar("x", 0, 10);
 IntegerVariable y = Choco.makeIntVar("y", 0, 10);
 model.addConstraint(Choco.reifiedConstraint(b, Choco.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 enforced on constraints in extension (when selecting an API) and, for Arc Consistency, different filtering algorithms can be used (when selecting an option) The Choco API for creating constraints in extension are as follows:

- arc consistency (AC) for binary relations:
  `feasPairAC`, `infeasPairAC`, `relationPairAC`
- arc consistency (AC) for n-ary relations:
  `feasTupleAC`, `infeasTupleAC`, `relationTupleAC`
- weaker forward checking (FC) for n-ary relations:
  `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 = Choco.makeLargeRelation(min, max, tuples, true);
    model.addConstraint(Choco.relationTupleAC(vars, r));
```

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

```
  public static 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, the `NotAllEqual` relation is stated as a constraint of a model:

```
        CPModel model = new CPModel();
        IntegerVariable[] vars = Choco.makeIntVarArray("v", 3, 1, 3);
        model.addConstraint(Choco.relationTupleFC(vars, new NotAllEqual()));
```

2.2.7 Reified constraints

The *truth value* of a constraint is a boolean that is true if and only if the constraint holds. To *reify* a constraint is to get its truth value.

This mechanism can be used for example to model a MaxCSP problem where the number of satisfied constraints has to be maximized. It is also intended to give the freedom to model complex constraints combining several reified constraints, using some logical operators on the truth values, such as in: $(x \neq y) \vee (z \leq 9)$.

Choco provides a generic constraint reifiedConstraint to reify any constraint into a boolean variable expressing its truth value:

```
  Constraint reifiedConstraint(IntegerVariable b, Constraint c);
  Constraint reifiedConstraint(IntegerVariable b, Constraint c1, Constraint c2);
```

Specific API are also provided to reify boolean constraints:

- `reifiedConstraint`,
- `reifiedAnd`, `reifiedLeftImp`, `reifiedNot`, `reifiedOr`, `reifiedRightImp`, `reifiedXnor`, `reifiedXor`

Handling complex expressions.

In order to build complex combinations of constraints, Choco also provides a more simple and direct API with the following logical meta-constraints taking constraints in arguments:

- `and`, `or`, `implies`, `ifOnlyIf`, `ifThenElse`, `not`, `nand`, `nor`

For example, the following expression

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

could be expressed in Choco by:

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

Such an expression is internally represented as a tree whose nodes are operators and leaves are variables, constants and 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 currently matches the language used in the Constraint Solver Competition 2008 of the CPAI workshop.

At the solver level, there exists two different ways to represent expressions:

- *by extension:* the first way is to handle expressions 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 achieved on the corresponding constraints.

- *by decomposition:* the second way is to decompose the expression automatically by introducing intermediate variables and eventually the generic `reifiedConstraint`. By doing so, the level of pruning decreases but expressions of larger arity involving large domains can be represented.

The way to represent expressions is decided at the modeling level. Representation *by extension* is the default. Representation *by decomposition* can be set instead by:

```
model.setDefaultExpressionDecomposition(true);
```

Representation *by decomposition* can also be decided individually for some expressions, by setting option `Options.E_DECOMP` when adding the constraint. For example, the following code tells the solver to decompose e1 but not e2 :

```
model.setDefaultExpressionDecomposition(false);
IntegerVariable[] x = makeIntVarArray("x", 3, 1, 3, Options.V_BOUND);

Constraint e1 = or(lt(x[0], x[1]), lt(x[1], x[0]));
model.addConstraint(Options.E_DECOMP, e1);

Constraint e2 = or(lt(x[1], x[2]), lt(x[2], x[1]));
model.addConstraint(e2);
```

When and how should I use expressions ?

Expressions offer a slightly richer modeling language than the one available via standard constraints. However, expressions can not be handled as efficiently as constraints that embed a dedicated propagation algorithm. We therefore recommend you to carefully check that you can not model the expression using the *global constraints* of Choco before using expressions.

Expressions represented *in extension* should be used in the case of complex logical relationships that involve **few different variables**, each of **small domain**, and if **arc consistency** is desired on those variables. In such a case, representation in extension can be much more effective than with decomposition. 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. Those constraints accept any number of variables and 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 listed below according to their application fields. Details and examples can be found in Section Elements of Choco/Constraints.

Value constraints

Constraints that put a restriction on how values can be distributed among a collection of variables. See also in Global Constraint Catalog: value constraint.

- `counting distinct values: allDifferent, atMostNValue, increasingNValue,`
- `counting values: among, occurrence, occurrenceMax, occurrenceMin, globalCardinality,`
- `indexing values: nth (element), max, min,`
- `ordering: sorting, increasingNValue, lex, lexeq, leximin, lexChain, lexChainEq,`
- `tuple matching: feasTupleAC, feasTupleFC, infeasTupleAC, infeasTupleFC, relationTupleAC, relationTupleFC,`
- `pattern matching: regular, costRegular, multiCostRegular, stretchPath, tree,`

Boolean constraints

Logical operations on boolean expressions. See also in Global Constraint Catalog: boolean constraint.

`and, or, clause,`

Channeling constraints

Constraints linking two collections of variables (many-to-many) or indexing one among many variables (one-to-many). See also Section Channeling and in Global Constraint Catalog: channelling constraint.

- `one-to-many: domainChanneling, nth (element), max, min,`
- `many-to-many: inverseChanneling, inverseSet, sorting, pack,`

Optimization constraints

Constraints channelling a variable to the sum of the weights of a collection of variable-value assignments. See also in Global Constraint Catalog: cost-filtering constraint.

- `one cost: among, occurrence, occurrenceMax, occurrenceMin, knapsackProblem, equation, costRegular, tree,`
- `several costs: globalCardinality, multiCostRegular,`

Packing constraints (capacitated resources)

Constraints involving items to be packed in bins without overlapping. More generaly, any constraints modelling the concurrent assignment of objects to one or several capacitated resources. See also in Global Constraint Catalog: resource constraint.

- `packing problems: equation, knapsackProblem, pack (bin-packing),`
- `geometric placement problems: geost,`
- `scheduling problems: disjoint (tasks) disjunctive, cumulative,`
- `timetabling problems: costRegular, multiCostRegular,`

Scheduling constraints (time assignment)

Constraints involving tasks to be scheduled over a time horizon. See also in Global Constraint Catalog: scheduling constraint.

- `temporal constraints: disjoint (tasks) precedence, precedenceDisjoint, precedenceImplied, precedenceReified, forbiddenInterval, tree,`
- `resource constraints: cumulative, disjunctive, geost,`

Chapter 3

The solver

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

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

This instruction creates a Constraint Programming (CP) `Solver` object.

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 two parts: variables reading and constraints reading.

3.1 Variables reading

The variables are declared in a model with a given type `IntegerVariable, SetVariable, RealVariable` and, possibly, with a given domain type (e.g. bounded or enumerated domains for integer and set variables). When reading the model, the solver iterates over the model variables, then creates the corresponding solver variables and domains data structures in accord with these types.

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-hard global constraint such as the cumulative constraint). Remember that switching from enumerated variables to bounded variables decreases the level of propagation achieved by the system.

Model variables and solver variables are distinct objects. Model variables implement the `Variable` interface and solver variables implement the `Var` interface. A model variable is defined by an abstract representation of its initial domain, while a solver variable encapsulates a concrete representation of the domain, and maintains its current state throughout the search. Hence, one cannot access to a variable value directly from a model variable but one can from its corresponding solver variable. The solver variables are anonymous but can be accessed from the corresponding model variables using the `Solver` API `getVar(Variable v)` and `getVar(Variable... v)`.

3.1.1 from IntegerVariable toIntDomainVar

For integer variables, the solver `IntDomainVar` is the counterpart to the model `IntegerVariable`. Methods `getVar(IntegerVariable var)` and `getVar(IntegerVariable... vars)` of `Solver` return the objects `IntDomainVar` and `IntDomainVar[]` respectively corresponding to `var` and `vars`:

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

The state of an `IntDomainVar` can be accessed using these main public methods:

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

The data structure representing the current domain within the `IntDomainVar` object depends on the domain type (bounded, enumerated, boolean, constant, etc.) of the model variable. See advanced uses for more informations on `IntDomainVar`.

3.1.2 from SetVariable to SetVar

For set variables, the solver `SetVar` is the counterpart to the model `SetVariable`. Methods `getVar(SetVariable var)` and `getVar(SetVariable... vars)` of `Solver` return the objects `SetVar` and `SetVar[]` respectively corresponding to `var` and `vars`:

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

Note that a set variable on integer values between 1 and n may have 2^n possible values, corresponding to every possible subsets of $\{1, 2, \ldots, n\}$. Hence, the domain of a `SetVar` is encoded by these bounds only: the lower bound, called the *kernel*, is the intersection of all possible set values, and the upper bound, called the *envelope*, is the union of all possible set values. Furthermore, a `SetVar` encapsulated an `IntDomainVar` representing the cardinality of the set variable. The domain type of this variable (enumerated or bounded) depends on the option given at the construction of the `SetVariable`.

The state of a `SetVar` can be accessed through these main public methods: Warning: Envelope is (french) spelled with two 'p' in the method name.

`SetVar` API	description
`getCard()`	returns the current cardinality (an `IntDomainVar` object)
`isInDomainKernel(int v)`	checks if value `v` belongs to the current kernel
`isInDomainEnveloppe(int v)`	checks if value `v` belongs to the current envelope
`getDomain()`	returns the current domain (a `SetDomain` object). Iterators on envelope or kernel can then be called
`getKernelDomainSize()`	returns the current size of the kernel
`getEnveloppeDomainSize()`	returns the current size of the envelope
`getEnveloppeInf()`	returns the current smallest value of the envelope
`getEnveloppeSup()`	returns the current largest value of the envelope
`getKernelInf()`	returns the current smallest value of the kernel
`getKernelSup()`	returns the current largest available value of the kernel
`getValue()`	returns the set value as a table of integers `int[]` when the variable is currently instantiated (kernel=envelope)

See advanced uses for more informations on `SetVar`.

3.1.3 from `RealVariable` to `RealVar`

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

For real variables, the solver `RealVar` is the counterpart to the model `RealVariable`. Methods `getVar(RealVariable var)` and `getVar(RealVariable... vars)` of `Solver` return the objects `RealVar` and `RealVar[]` respectively corresponding to `var` and `vars`:

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

Continuous variables are useful for non linear equation systems which are encountered in physics for example. The state of a `RealVar` can be accessed through these main public methods:

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

See advanced uses for more informations on `RealVar`.

3.1.4 from `TaskVariable` to `TaskVar`

For task variables, the solver `TaskVar` is the counterpart to the model `TaskVariable`. Methods `getVar(TaskVariable var)` and `getVar(TaskVariable... vars)` of `Solver` return the objects `TaskVar` and `TaskVar[]` respectively corresponding to `var` and `vars`:

```
TaskVariable x = Choco.makeTaskVar("x", 0, 123, 18); // model variable
TaskVar xOnSolver = solver.getVar(x); // solver variable
```

Task variables help at formulating scheduling problems where one have to determine the starting and ending time of a task. A task variable aggregates three integer variable: `start`, `end`, and `duration` and the implicit constraint `start`+`duration`=`end`.

To complete. The state of a `TaskVar` can be accessed through these main public methods:

`TaskVar` API	description
`isInstantiated()`	checks if the three time integer variables are instantiated

See advanced uses for more informations on `TaskVar`.

3.2 Constraints reading

Once the solver variables are created when reading the model, the solver then iterates over the constraints of the model, and creates the solver `SConstraint` objects by calling method `makeConstraint` of the `ConstraintManager` object associated to the model constraint type. At this step, auxiliary solver variables and constraints may be generated. The created constraints are then added to the internal constraint network.

Each solver constraint encapsulates a filtering algorithm which is called, during the search, when a propagation step occurs or when an external event (e.g., value removal or bound modification) happens on some variable of the constraint.

3.3 Search Strategy

A key ingredient of any constraint approach is a clever search strategy. In backtracking or branch-and-bound approaches, the search is organized as an enumeration tree, where each node corresponds to a subspace of the search, and each child node is a subdivision of the space of its father node. The tree is progressively constructed by applying a series of branching strategies that determine how to subdivise space at each node and in which order to explore the created child nodes. Branching strategies play the role of achieving intermediate goals in logic programming.

This section presents how to define your own search strategy in Choco.

Standard backtracking or branch-and-bound approaches in constraint programming develop the enumeration tree in a **Depth-First Search (DFS)** manner:

1. *evaluate* a node: run propagation
2. if a failure occurs or if the search space cannot be separated then *backtrack*: evaluate the next pending node
3. otherwise *branch*: divide the search space and evaluate the first child node.

With Choco, the search process of the `CPSolver` does not currently allow to explore the tree in a different manner, using Best-First Search for example.
In addition, the common way of dividing the search space in CP-based backtracking/B&B algorithms is **to assign a variable to a value or to forbid this assignment**. Choco provides such a branching strategy and the tools to easily customize the variable and value selection heuristics within this strategy. However, Choco makes possible to implement **more complex branching strategies** (e.g. constraint branching or dichotomy branching).

3.3.1 Overriding the default search strategy

Branching, variable selection and value selection strategies. Basically, a search strategy in Choco is a composition of branching strategy objects, each defined on a given set of decision variables. The most common branching strategies are based on the assignment of a selected variable to one or several selected values (one assignment in each branch).

Branching strategies apply to `Solver` variables (not `Model` variables).

The variable and value selection heuristics can be defined separately in their own objects: a variable selector and a value selector or a value iterator. Branching strategy `AssignVar`, for example, can simply be customized via these embedded objects: the variable is first selected, then a value in the variable domain is selected. The two following instructions both create a n-ary branching strategy (`AssignVar`) selecting an integer decision variable of minimum domain size (`MinDomain` variable selector) and assigning it successively, in each branch, to one of its domain value, selected in increasing order (`IncreasingDomain` value iterator or `MinVal` value selector).

```
new AssignVar(new MinDomain(solver), new IncreasingDomain());
new AssignVar(new MinDomain(solver), new MinVal());
```

Note that this usual strategy is pre-defined in `BranchingFactory`, and may then also be declared as follows:

```
BranchingFactory.minDomMinVal(solver);
```

Sometimes, the choice of the variable may also depend on the choice of the value, or it may require specific computations before or after branching. In this case, the variable selection heuristic can directly be implemented in the branching strategy object, like e.g. in DomOverWDegBranchingNew. Both variable and value selection heuristics can be implemented directly within the branching strategy, like e.g in ImpactBasedBranching.

Default strategies. When no search strategy is specified, then default search strategies apply to all the decision variables of the solver. These strategies vary in function of the variable types:

Variable type	Default strategy
Set	AssignSetVar + MinDomainSet + MinEnv
Integer	DomOverWDegBranchingNew +IncreasingDomain
Real	AssignInterval + CyclicRealVarSelector+ RealIncreasingDomain

If the model has decision variables of different types, then these default branchings are evaluated in this order: first, the set decision variables are considered until they are all instantiated, then branching occurs on the pool of integer decision variables, and last on the pool of real decision variables.

Decision variables. Branchings apply to decision variables only. A branching can occur (i.e. the tree node can be separated according to this strategy) if and only if there exists a decision variable in its scope that is still not instantiated. The non-decision variables are also called *implied variables* because it is expected that, all variables – including these – will be instantiated (i.e. they will form a solution) by propagation as soon as all the decision variables will be all instantiated. Consider for example, a problem with two sets of variables x and y linked by channeling or by some implication $x = S \implies y = T$ then the x variables can be set as the decision variables, while the y variables can be let implied.

By default, every solver variable belongs to the pool of decision variables, unless:

- it corresponds to a model variable created with flag `Options.V_NO_DECISION` set
- or it is internally created by the solver (e.g. when reading some model constraint) and explicitely excluded from the pool
- or the default branching strategies are overriden and the variable does not belong to the scope of one of the strategies specified by the user.

The scope of a branching strategy is defined at the creation of the strategy. For example,

```
new AssignSetVar(new MinDomSet(solver, solver.getVar(svars)), new MinEnv());
```

defines a branching strategy that only applies to the solver variables corresponding to the model set variables `svars` (even if they were defined with flag `Options.V_NO_DECISION`).

Most branching strategies may be declared without specifying their scope. In this case, they apply to all the solver decision variables of the right type. For example, the branching strategy

```
new AssignSetVar(new MinDomSet(solver), new MinEnv());
```

now applies to all the solver decision set variables.

If the default strategies of the solver are overridden by this strategy alone, then all other integer and real variables will automatically be removed from the decision pool: one has then to ensure that the instantiation of the set variables alone defines a complete solution. If it is not the case, the branching strategy must be combined with additional branching strategies holding on the remaining unimplied variables.

> If the default strategies are overriden, then the pool of decision variables is overriden by the union of the scopes of the user-specified branching strategies.

As the branching strategies are evaluated sequentially, a variable may belong to the scope of two different strategies, but it will only be considered by the first strategy, unless this first (user-defined) strategy let the variable un-instantiated.

Overriding the default search strategies. A branching strategy is added to the solver, as a goal, using the following API of `Solver`:

```
void addGoal(AbstractIntBranchingStrategy branching);
```

This method must be called on the solver object *before* calling the solving method. The initial list of goals is empty. If goals are specified, they are added to the list in the order of their declaration. Otherwise, the list is initialized with the default goals (in the order: set, integer, real).

When one wants to relaunch the search, the list of goals of the solver can previously be reset using the following instruction:

```
solver.clearGoals();
```

Complete example. The following example adds four branching objects to solver `s`. The first two branchings are both `AssignVar` strategies using different variable/value selection heuristics and applied to different scopes: the integer variables `vars1` and `vars2`, respectively. The third strategy applies to the set variables `svars`. The last random strategy applies to all the integer decision variables of the solver.

```
s.addGoal(BranchingFactory.minDomMinVal(s, s.getVar(vars1)));
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.addGoal(BranchingFactory.randomIntSearch(s, seed));
s.solve();
```

The goals are evaluated in this order: first, variables `vars1` are considered until they are all instantiated, then branching occurs on variables `vars2`, then on variables `svars`. Finally, a random strategy is applied to all the integer decision variables of the solver that are not really instantiated, a fortiori, excluding variables `vars1` and `vars2`.

3.3.2 Pre-defined search strategies

This section presents the strategies available in Choco. These objects are also detailed in Part Elements of Choco. See Chapter advanced uses for a description of how to write search strategies in Choco.

Branching strategy defines the way to branch from a tree search node.
The **branching strategies** currently available in Choco are the following:

`AssignInterval`, `AssignOrForbidIntVarVal`, `AssignOrForbidIntVarValPair`, `AssignSetVar`, `AssignVar`, `DomOverWDegBranchingNew`, `DomOverWDegBinBranchingNew`, `ImpactBasedBranching`, `PackDynRemovals`, `SetTimes`, `TaskOverWDegBinBranching`.

They implement interface `BranchingStrategy`.

Variable selector defines the way to choose a non instantiated variable on which the next decision will be made.
The **variable selectors** currently available in Choco are the following:

- implementing interface `VarSelector<IntDomainVar>`:

  `CompositeIntVarSelector`, `LexIntVarSelector`, `MaxDomain`, `MaxRegret`, `MaxValueDomain`, `MinDomain`, `MinValueDomain`, `MostConstrained`, `RandomIntVarSelector`, `StaticVarOrder`

- implementing interface `VarSelector<SetVar>`:

```
MaxDomSet, MaxRegretSet, MaxValueDomSet, MinDomSet, MinValueDomSet,
MostConstrainedSet, RandomSetVarSelector, StaticSetVarOrder
```

- implementing interface `VarSelector<RealVar>`:

```
CyclicRealVarSelector
```

Value iterator

Once the variable has been choosen, the solver has to compute its value. The first way to do it is to schedule all the values once and to give an iterator to the solver.
The **value iterators** currently available in Choco are the following:

- implementing interface `ValIterator<IntDomainVar>`:

```
DecreasingDomain, IncreasingDomain
```

- implementing interface `ValIterator<RealVar>`:

```
RealIncreasingDomain
```

Value selector

The second way to do it is to compute the next value at each call.
The **integer value selector** currently available in Choco are the following:

- implementing interface `ValSelector<IntDomainVar>`:

```
  – MaxVal, MidVal, MinVal
  – BestFit, CostRegularValSelector, FCostRegularValSelector,
```

- implementing interface `ValSelector<SetVar>`:

```
MinEnv, RandomSetValSelector
```

3.3.3 Why is it important to define a search strategy ?

> *The search strategy should not be overlooked!!* A suited search strategy can reduce: the execution time, the number of expanded nodes, the number of backtracks.

Let see that small example:

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

     Solver badStrat = new CPSolver();
     badStrat.read(m);
     badStrat.addGoal(
            new AssignVar(
                   new MinDomain(badStrat),
                   new IncreasingDomain()
            ));
     badStrat.solve();
     badStrat.printRuntimeStatistics();

     Solver goodStrat = new CPSolver();
     goodStrat.read(m);
     goodStrat.addGoal(
            new AssignVar(
                   new MinDomain(goodStrat, goodStrat.getVar(new IntegerVariable[]{var})),
                   new IncreasingDomain()
            ));
     goodStrat.solve();
     goodStrat.printRuntimeStatistics();
```

This model ensures that $var = b_0 + b_1 + \ldots + b_{1000}$ where var is an integer variable with a small domain $[0, 2]$ and b_i are binary variables. No deduction arose from the propagation here, so a fix point is reached at the beginning of the search. A branching decision has to be taken, by selecting a variable and the first value to assign to it. Using the first strategy, the solver will find a solution after creating 1001 nodes: it iterates over all the variables, starting by assigning the 1000 binary variables b_i (according to the `MinDomain` variable selector) to 0 (according to the `IncreasingDomain` value iterator), variable var is fixed to 0 at the very last propagation. The second strategy finds the same solution with only two nodes: after branching first on $var = 0$, propagation immediately fixes all the binary variables to 0.

3.3.4 Restarts

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

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

Restarts can be set 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 backtracks. Parameter `base` indicates the maximal number of backtracks allowed in the first search tree. Once this limit is reached, a restart occurs and the search continues until `base*grow` backtracks are done, and so on. After each restart, the limit number of backtracks is increased by the geometric factor `grow`. `restartLimit` states the maximum number of restarts.

```
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();
```

The Luby's restart policy is an alternative to the geometric restart policy, and can be defined 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. The maximum number of backtracks allowed at a given restart iteration is given by `base` multiplied by the Las Vegas coefficient at this iteration. The sequence of these coefficients is defined recursively on its prefix subsequences: starting from the first prefix 1, the $(k+1)$-th prefix is the k-th prefix repeated `grow` times and immediately followed by coefficient $\texttt{grow}^k$.

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

```
  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 ways to limit the tree search regarding different criteria. These limits have to be specified before the resolution. They are updated and checked each time a new node is created. Once a limit is reached, the search stops even if no solution is found.

time limit: concerns the ellapsed time from the beginning of the search (i.e. from the call to a resolution method). A time limit is set using the `Solver` API `setTimeLimit(int timeLimit)`, where unit is millisecond. `getTimeCount()` returns the total solving time.

node limit: concerns the number of explored nodes. A node limit is set using the `Solver` API: `setNodeLimit(int nodeLimit)`. `getNodeCount()` returns the total number of explored nodes.

backtrack limit: concerns the number of performed backtracks A backtrack limit is set using the `Solver` API: `setBackTrackLimit(int backtrackLimit)`. `getBackTrackCount()` return the total number of backtracks.

fail limit: concerns the number of failures. A fail limit is set using the `Solver` API : `setFailLimit(int failLimit)`. where unit is the number of failure. Or just monitor (or not) t The number of failures can be monitored using `monitorFailLimit(true)`. `getFailCount()` returns the total number of failures.

Define all these notions more precisely and add an example.

3.5 Solve a problem

Table below presents the different API offers by `Solver` to launch the problem resolution. All these methods return a `Boolean` object standing for the *problem feasibility status* of the solver:

$$\begin{cases} \texttt{Boolean.TRUE} & \text{if at least one feasible solution has been computed,} \\ \texttt{Boolean.FALSE} & \text{if the problem is proved to be infeasible,} \\ \texttt{null} & \text{otherwise, i.e. when a search limit has been reached before.} \end{cases}$$

`Solver` API	description
`solve()` or `solve(false)`	runs backtracking until reaching *a first feasible solution* (returns `Boolean.TRUE`) or the proof of infeasibility (returns `Boolean.FALSE`) or a search limit (returns `null`).
`solveAll()` or `solve(true)`	Runs backtracking until computing *all feasible solutions*, or until proving infeasibility (returns `Boolean.FALSE`) or until reaching a search limit (returns `Boolean.TRUE` if at least one first solution was computed, and `null` otherwise).
`maximize(Var obj, boolean restart)`	Runs branch-and-bound until reaching *a feasible solution that is proved to maximize objective* `obj`, or until proving infeasibility (returns `Boolean.FALSE`) or until reaching a search limit (returns `Boolean.TRUE` if at least one first solution was computed, and `null` otherwise). It proceeds by successive improvements of the best solution found so far: each time a feasible solution is found at a leaf of the tree search, then the search follows for a new solution with a greater objective, until it proves that no such improving solution exists. Parameter `restart` is a boolean indicating whether the search continues from the solution leaf with a backtrack (if set to `false`) or if it is relaunch from the root node (if set to `true`). See example.
`minimize(Var obj, boolean restart)`	similar to `maximize` but for computing *a feasible solution that is proved to minimize objective* `obj`.
`nextSolution()`	Can only be called after a `solve()` or a `nextSolution()` call that has returned `Boolean.TRUE`. Runs backtracking, from the solution leaf reached by the previous `solve()` or `nextSolution()` call, until reaching *a new feasible solution* (returns `Boolean.TRUE`), or proving no such new solution exists (returns `Boolean.FALSE`), or reaching a search limit (returns `null`).
`isFeasible()`	Returns the feasibility status of the solver.

The following API are also useful to manipulate a `Solver` object:

`Solver` API	description
`propagate()`	Launchs the initial propagation by running, in turn, the domain reduction algorithms of the constraints until it reaches a fix point. Throws a `ContradictionException` when a contradiction is detected, i.e. a variable domain is emptied.
`isFeasible()`	Returns the current feasibility status of the solver.

3.5.1 Solver settings

Logs

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

```
ChocoLogging.toVerbose();
// And after solver.solve()
ChocoLogging.flushLogs();
```

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

Six verbosity levels are available:

Level	prints...
`ChocoLogging.toSilent()`	display only severe messages from core loggers and warning messages otherwise
`ChocoLogging.toQuiet()`	display only severe messages from core loggers and info messages otherwise
`ChocoLogging.toDefault()`	display information about initial and final state of the search
`ChocoLogging.toVerbose()`	display search information at regular node intervals
`ChocoLogging.toSolution()`	display all solutions
`ChocoLogging.toSearch()`	display the search tree

Note that in the case of a verbosity greater or equals to `toVerbose()`, the regular search information step is set to 1000, by default. You can change this value, using:

```
ChocoLogging.setEveryXNodes(20000);
```

Note that in the case of verbosity `toSearch()`, trace statements are printed up to a maximal depth in the search tree. The default value is set to 25, but you can change the value of this threshold, say to 10, with the following setter method:

```
ChocoLogging.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 Environment

Environment is a central object of the backtracking system. It defines the notion of *world*. A world contains values of storable objects or operations that permit to *backtrack* to its state. The environment *pushes* and *pops* worlds.

There are *primitive* data types (`IstateBitSet, IStateBool, IStateDouble, IStateInt, IStateLong`) and *objects* data types (`IStateBinarytree, IStateIntInterval, IStateIntProcedure, IStateIntVector, IStateObject, IStateVector`).

There are two different environments: *EnvironmentTrailing* and *EnvironmentCopying*.

4.1.1 Copying

In that environment, each data type is defined by a value (primitive or object) and a timestamp. Every time a world is pushed, each value is copied in an array (one array per data type), with finite indice. When a world is popped, every value is restored.

4.1.2 Trailing

In that environment, data types are defined by its value. Every operation applied to a data type is pushed in a *trailer*. When a world is pushed, the indice of the last operation is stored. When a world is popped, these operations are popped and *unapplied* until reaching the last operation of the previous world.
Default one in CPSolver

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:

- `AbstractIntBranchingStrategy`: abstract class for the branching strategy,
- `VarSelector<V>` : Interface for the variable selection (V extends Var),
- `ValIterator<V>` : Interface to describes an iteration scheme on the domain of a variable,
- `ValSelector<V>` : Interface for a value selection.

Concrete examples of these interfaces are respectively, AssignVar, MinDomain, IncreasingDomain, MaxVal.

4.2.1 How to define your own Branching object

Beyond Variable/value selection...

4.2.2 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 as a part of a search strategy, using `solver.addGoal()`.

4.2.3 Define your own value selection

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

Value selector

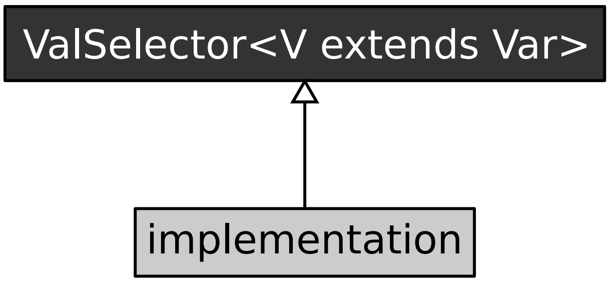


Figure 4.1: ValSelector interface and implementation

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 as a part of a search strategy, using `solver.addGoal()`.

> Using a value selector with bounded domain variable is strongly inadvised, except if it pick up bounds value. If the value selector pick up a value that is not a bound, when it goes up in the tree search, that value could be not removed and picked twice (or more)!

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 as a part of a search strategy, using `solver.addGoal()`.

under development See old version

4.3 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();
```

4.3.1 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
```

4.4 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)

4.4.1 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 re-propagation.
`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.

4.4.2 Example: implement and add the `IsOdd` constraint

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

```
public class IsOdd extends AbstractUnIntSConstraint {

   @Override
   public int getFilteredEventMask(int idx) {
      return IntVarEvent.INSTINTbitvector;
   }

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

      /**
       * Default initial propagation: full constraint re-propagation.
       */

      public void awake() throws ContradictionException {
         DisposableIntIterator it = v0.getDomain().getIterator();
         try{
            while(it.hasNext()){
               int val = it.next();
               if(val%2==0){
                  v0.removeVal(val, cIdx0);
               }
            }
         }finally {
            it.dispose();
         }
      }

      /**
       * <i>Propagation:</i>
```

```
        * Propagating the constraint until local consistency is reached.
        *
        * @throws ContradictionException
        * contradiction exception
        */

       public void propagate() throws ContradictionException {
           if(v0.isInstantiated()){
               if(v0.getVal()%2==0){
                   fail();
               }
           }
       }


   }
```

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

```
public class IsOddManager extends IntConstraintManager {
    public SConstraint makeConstraint(Solver solver, IntegerVariable[] variables, Object
         parameters, Set<String> options) {
        if (solver instanceof CPSolver) {
            return new IsOdd(solver.getVar(variables[0]));
        }
        return null;
    }
}
```

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!!

4.4.3 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.

4.5 Define your own operator

to complete

4.6 Define your own variable

to complete

4.7 Backtrackable structures

to complete

4.8 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.

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.

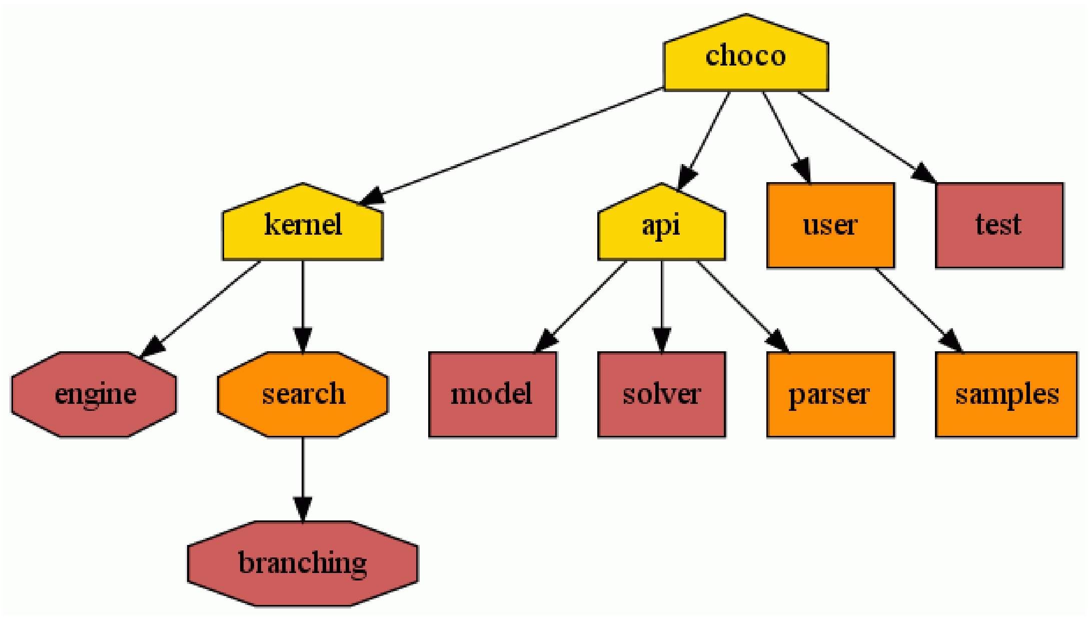


Figure 4.2: Logger Tree with the default verbosity

- 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 and messages.

The following table summarizes the verbosities available in choco:

- **OFF** – **level 0:** Disable logging.
- **SILENT** – **level 1:** Display only severe messages.
- **DEFAULT** – **level 2:** Display informations on final search state.
  - ON START
    ```
    ** CHOCO : Constraint Programming Solver
    ** CHOCO v2.1.1 (April, 2009), Copyleft (c) 1999-2010
    ```
  - ON COMPLETE SEARCH:
    ```
    - Search completed -
     [Maximize      : {0},]
     [Minimize      : {1},]
      Solutions     : {2},
      Times (ms)    : {3},
      Nodes         : {4},
      Backtracks    : {5},
      Restarts      : {6}.
    ```
    brackets [*line*] indicate *line* is optionnal,
    `Maximize` –resp. `Minimize`– indicates the best known value before exiting of the objective value in *maximize()* – –resp. *minimize()*– strategy.
  - ON COMPLETE SEARCH WITHOUT SOLUTIONS :

```
- Search completed - No solutions
 [Maximize      : {0},]
 [Minimize      : {1},]
  Solutions     : {2},
  Times (ms)    : {3},
  Nodes         : {4},
  Backtracks    : {5},
  Restarts      : {6}.
```

brackets [*line*] indicate *line* is optionnal,
`Maximize` –resp. `Minimize`– indicates the best known value before exiting of the objective value in *maximize()* – –resp. *minimize()*– strategy.

– ON INCOMPLETE SEARCH:

```
- Search incompleted - Exiting on limit reached
  Limit         : {0},
 [Maximize      : {1},]
 [Minimize      : {2},]
  Solutions     : {3},
  Times (ms)    : {4},
  Nodes         : {5},
  Backtracks    : {6},
  Restarts      : {7}.
```

brackets [*line*] indicate *line* is optionnal,
`Maximize` –resp. `Minimize`– indicates the best known value before exiting of the objective value in *maximize()* – –resp. *minimize()*– strategy.

- **VERBOSE – level 3:** Display informations on search state.

  – EVERY X (default=1000) NODES:

```
- Partial search - [Objective : {0}, ]{1} solutions, {2} Time (ms), {3} Nodes, {4}
    Backtracks, {5} Restarts.
```

`Objective` indicates the best known value.

  – ON RESTART :

```
- Restarting search - {0} Restarts.
```

- **SOLUTION – level 4:** display all solutions.

  – AT EACH SOLUTION:

```
- Solution #{0} found. [Objective: {0}, ]{1} Solutions, {2} Time (ms), {3} Nodes, {4}
     Backtracks, {5} Restarts.
  X_1:v1, x_2:v2...
```

- **SEARCH – level 5:** Display the search tree.

  – AT EACH NODE, ON DOWN BRANCH:

```
...[w] down branch X==v branch b
```

where `w` is the current world index, `X` the branching variable, `v` the branching value and `b` the branch index. This message can be adapted on variable type and search strategy.

  – AT EACH NODE, ON UP BRANCH:

```
...[w] up branch X==v branch b
```

where `w` is the current world index, `X` the branching variable, `v` the branching value and `b` the branch index. This message can be adapted on variable type and search strategy.

- **FINEST – level 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);
```

Part II

Elements of Choco

Chapter 5

Variables (Model)

This section describes the three kinds of variables that can be used within a Choco Model, and an object-variable.

5.1 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 `Options.V_ENUM`
- `Options.V_ENUM` : to force Solver to create enumerated domain for the variable.
- `Options.V_BOUND` : to force Solver to create bounded domain for the variable.
- `Options.V_LINK` : to force Solver to create linked list domain for the variable.
- `Options.V_BTREE` : to force Solver to create binary tree domain for the variable.
- `Options.V_BLIST` : to force Solver to create bipartite list domain for the variable.
- `Options.V_MAKEPSAN` : declare the current variable as makespan.
- `Options.V_NO_DECISION` : to force variable to be removed from the pool of decisional variables.
- `Options.V_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, Options.V_BOUND, Options.
        V_NO_DECISION);
    IntegerVariable bool = makeBooleanVar("bool");
```

Integer variables are illustrated on the n-Queens problem.

5.2 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):

```
    Solver m = new CPSolver();
    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.
- `Options.V_NO_DECISION` : to force variable to be removed from the pool of decisional variables.
- `Options.V_OBJECTIVE` : to define the variable to be the one to optimize.

Example:

```
    RealVariable rvar1 = makeRealVar("rvar1", -10.0, 10.0);
    RealVariable rvar2 = makeRealVar("rvar2", 0.0, 100.0, Options.V_NO_DECISION, Options.
        V_OBJECTIVE);
```

Real variables are illustrated on the CycloHexan problem.

5.3 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 `Options.V_ENUM`
- `Options.V_ENUM` : to force Solver to create `SetVariable` with enumerated domain for the caridinality variable.
- `Options.V_BOUND` : to force Solver to create `SetVariable` with bounded cardinality.
- `Options.V_NO_DECISION` : to force variable to be removed from the pool of decisional variables.
- `Options.V_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, Options.V_BOUND, Options.V_NO_DECISION
    );
```

Set variables are illustrated on the ternary Steiner problem.

5.4 Task variables

`TaskVariable` is an object-variable composed of three `IntegerVariable`: a starting time integer variable *start*, an ending time integer variable *end* and a duration integer variable *duration*. To create a `TaskVariable`, one can creates the *start*, *end* and *duration* before, or indicates the earliest starting time (*int*), the latest completion time (*int*) and the duration(`int` or `IntegerVariable`).

constructors:

Choco method	return type
`makeTaskVar(String name, IntegerVariable start, IntegerVariable end, IntegerVariable duration, String... options)`	`TaskVariable`
`makeTaskVar(String name, IntegerVariable start, IntegerVariable duration, String... options)`	`TaskVariable`
`makeTaskVar(String name, int binf, int bsup, IntegerVariable duration, String... options)`	`TaskVariable`
`makeTaskVar(String name, int binf, int bsup, int duration, String... options )`	`TaskVariable`
`makeTaskVar(String name, int bsup, IntegerVariable duration, String... options)`	`TaskVariable`
`makeTaskVar(String name, int bsup, int duration, String... options)`	`TaskVariable`
`makeTaskVarArray(String prefix, IntegerVariable[] starts, IntegerVariable[] ends, IntegerVariable[] durations, String... options)`	`TaskVariable[]`
`makeTaskVarArray(String name, int binf, int bsup, IntegerVariable[] durations, String... options)`	`TaskVariable[]`
`makeTaskVarArray(String name, int binf, int bsup, int[] durations, String... options)`	`TaskVariable[][]`
`makeTaskVarArray(String name, int binf, int bsup, IntegerVariable[][] durations, String... options)`	`TaskVariable[][]`
`makeTaskVarArray(String name, int binf, int bsup, int[][] durations, String ... options)`	`TaskVariable[][]`

options:

Options are for the three `IntegerVariable`. See `IntegerVariable` for more details about options.

Example:

```
    TaskVariable tvar1 = makeTaskVar("tvar1", 0, 123, 18, Options.V_ENUM);
    IntegerVariable start = makeIntVar("start", 0, 30);
    IntegerVariable end = makeIntVar("end", 10, 60);
    IntegerVariable duration = makeIntVar("duration", 7, 13);
    TaskVariable tvar2 = makeTaskVar("tvar2", start, end, duration);
```

Chapter 6

Operators (Model)

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

6.1 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, Options.V_ENUM);
IntegerVariable y = makeIntVar("y", -5, 5, Options.V_ENUM);
m.addConstraint(eq(abs(x), y));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

6.2 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:

```
Model m = new CPModel();
RealVariable x = makeRealVar("x", -Math.PI/2, Math.PI);
m.addConstraint(eq(cos(x), 2/3));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

6.3 distEq (operator)

To complete

6.4 distGt (operator)

To complete

6.5 distLt (operator)

To complete

6.6 distNeq (operator)

To complete

6.7 div (operator)

Returns an expression variable that represents the *integer quotient* 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();
```

6.8 ifThenElse (operator)

`ifThenElse`(c, v_1, v_2) states that if the constraint c is satisfied, it returns the second parameter v_1, otherwise it returns the third one v_2.

- **API** : `ifThenElse(Constraint c, IntegerExpressionVariable v1, IntegerExpressionVariable v2)`
- **return type** : IntegerExpressionVariable
- **options** : *n/a*
- **favorite domain** : unknown

Example:

```
Model m = new CPModel();
IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 0, 10);
m.addConstraint(eq(y, ifThenElse(gt(x,2), mult(x,x), x)));
Solver s = new CPSolver();
s.read(m);
s.solveAll();
```

6.9 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();
```

6.10 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();
```

6.11 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();
```

6.12 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();
```

6.13 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();
```

6.14 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();
```

6.15 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();
```

6.16 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();
```

6.17 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();
```

6.18 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:

```
Model m = new CPModel();
RealVariable x = makeRealVar("x", 0, Math.PI);
m.addConstraint(eq(sin(x), 1));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

6.19 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();
}
```

Chapter 7

Constraints (Model)

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

7.1 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, Options.V_ENUM);
    IntegerVariable y = makeIntVar("y", -5, 5, Options.V_ENUM);
    m.addConstraint(abs(x, y));
    Solver s = new CPSolver();
    s.read(m);
    s.solve();
```

7.2 allDifferent (constraint)

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

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

This constraint is the basis of any matching problems. Notice that the filtering algorithm (AC [Régin, 1994] or BC [López-Ortiz et al., 2003]) depends on the nature (enumerated or bounded) of variables x.

- **API** :

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

- **return type** : `Constraint`
- **options** :
  - *no option*: if the domains of x are *enumerated*, the constraint refers to the alldifferent of [Régin, 1994]; if they are *bounded*, a dedicated algorithm [López-Ortiz et al., 2003] for bound propagation is used instead.
  - `Options.C_ALLDIFFERENT_AC` for [Régin, 1994] implementation of arc consistency
  - `Options.C_ALLDIFFERENT_BC` for [López-Ortiz et al., 2003] implementation of bound consistency
  - `Options.C_ALLDIFFERENT_CLIQUE` for propagating the clique of differences
- **favorite domain** : *enumerated* for arc consistency, *bounded* for bound consistency.
- **references** :
  - [Régin, 1994]: *A filtering algorithm for constraints of difference in CSPs*
  - [López-Ortiz et al., 2003]: *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(Options.C_ALLDIFFERENT_CLIQUE, allDifferent(diag1));
m.addConstraint(Options.C_ALLDIFFERENT_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());
```

7.3 among (constraint)

`among`$(z, \langle x_1, .., x_n \rangle, s)$ states that z is the number of x_i belonging to set s:

$$z = |\{i \mid x_i \in s\}|$$

- **API**:
  - `among(IntegerVariable z, IntegerVariable[] x, int[] v)`
  - `among(IntegerVariable z, IntegerVariable[] x, SetVariable s)`
- **return type**: `Constraint`
- [Bessière et al., 2005a]: *Among, common and disjoint Constraints*
- [Bessière et al., 2006]: *Among, common and disjoint Constraints*
- global constraint catalog: `among`

Example:
among with a collection of values

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable nvar = makeIntVar("v1", 1, 2);
IntegerVariable[] vars = Choco.makeIntVarArray("var", 10, 0, 10);
int[] values = new int[]{2, 3, 5};
m.addConstraint(among(nvar, vars, values));
s.read(m);
s.solve();
```

among with a set variable

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable nvar = makeIntVar("v1", 1, 2);
IntegerVariable[] vars = Choco.makeIntVarArray("var", 10, 0, 10);
SetVariable values = Choco.makeSetVar("s", 2, 6);
m.addConstraint(among(nvar, vars, values));
s.read(m);
s.solve();
```

7.4 and (constraint)

`and`$(\langle C_1, .., C_n\rangle)$ states that constraints in arguments are all satisfied:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

`and`$(\langle b_1, .., b_n\rangle)$ states that booleans in arguments are all true:

$$(b_1 = 1) \;\wedge\; (b_2 = 1) \;\wedge\; \ldots \;\wedge\; (b_n = 1)$$

- **API** :
  - `and(Constraint... c)`
  - `and(IntegerVariable... b)`
- **return type** : `Constraint`
- **options** : *n/a*

- **favorite domain** : *n/a*
- **references** :
  global constraint catalog: `and`

Examples:

- example1:

```
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();
```

- example2

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable[] vars = makeBooleanVarArray("b", 10);
m.addConstraint(and(vars));
s.read(m);
s.solve();
```

7.5 atMostNValue (constraint)

`atMostNValue`$(z, \langle x_1, .., x_n\rangle)$ states that the number of distinct values occurring in collection x is at most z:

$$z \geq |\langle x_1, .., x_n\rangle|$$

- **API** : `atMostNValue(IntegerVariable z, IntegerVariable[] x)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*
- **references** :
  - [Bessière et al., 2005b] *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(n, new IntegerVariable[]{v1, v2, v3, v4});
```

```
        m.addConstraint(c);
        s.read(m);
        s.solve();
```

7.6 boolChanneling (constraint)

`boolChanneling`(b, x, v) states that boolean b is true if and only if x has value v:

$$(b = 1) \iff (x = v)$$

b is an indicator variable acting as an observer of value v. See also `domainChanneling` for observing all the values of x.

- **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();
```

7.7 clause (constraint)

`clause`$(\langle b_1^+, .., b_n^+\rangle, \langle b_1^-, .., b_m^-\rangle)$ states that at least one boolean b_i^+ is true or one boolean b_j^- is false.

$$\bigvee_{i=1}^{n} (b_i^+ = 1) \vee \bigvee_{j=1}^{m} (b_j^- = 0)$$

- **API** :
  - `clause(IntegerVariable[] bpos, IntegerVariable[] bneg)`
  - `clause(String options, IntegerVariable[] bpos, IntegerVariable[] bneg)`
- **return type** : `Constraint`
- **options** :
  - *no option* default filtering
  - `Options.C_CLAUSES_ENTAIL` ensures quick entailment tests

- **favorite domain** : *n/a.*
- **references** : global constraint catalog: clause_or

Example:

```
CPModel mod = new CPModel();
CPSolver s = new CPSolver();
IntegerVariable[] vars = makeBooleanVarArray("b", 8);

IntegerVariable[] plits1 = new IntegerVariable[]{vars[0], vars[3], vars[4]};
IntegerVariable[] nlits1 = new IntegerVariable[]{vars[1], vars[2], vars[6]};
mod.addConstraint(clause(plits1, nlits1));

IntegerVariable[] plits2 = new IntegerVariable[]{vars[5], vars[3]};
IntegerVariable[] nlits2 = new IntegerVariable[]{vars[1], vars[4], vars[7]};
mod.addConstraint(clause(plits2, nlits2));

s.read(mod);
s.solveAll();
```

7.8 costRegular (constraint)

`costRegular`$(z, \langle x_1, .., x_n\rangle, \mathcal{L}(\Pi), \langle c_{i,j}\rangle)$ states that sequence $\langle x_1, .., x_n\rangle$ is a word belonging to the regular language $\mathcal{L}(\Pi)$ and that z is its cost computed as the sum of the individual symbol weights c_{i,x_i}:

$$\langle x_1, .., x_n\rangle \in \mathcal{L}(\Pi) \quad \wedge \quad \sum_{i=1}^{n} c_{i,x_i} = z.$$

Like regular, this constraint is useful for modelling sequencing rules in personnel scheduling and rostering problems. Furthermore it allows to handle a linear counter (or cost) on the sequence. See multiCostRegular to simultaneously handle several linear counters.

`costRegular` is the optimization variant of the regular constraint. Enforcing GAC is NP-Hard, then the implemented algorithm [Demassey et al., 2006] achieves an intermediate AC-BC level of consistency. Let $\mathcal{L}_x = \mathcal{L}(\Pi) \cap (D_1 \times \ldots \times D_n)$ be the set of words of the language $\mathcal{L}(\Pi)$ that can be matched by $\langle x_1, .., x_n\rangle$ according to their current domains $\langle D_1, .., D_n\rangle$, then:

- Arc Consistency is enforced over x regarding the language and the lower and upper bounds of z: for each value $v \in D_i$, there exists a word in $\mathcal{L}_x$, with v as its i-th symbol and whose cost is between the bounds of z.
- Bound Consistency is enforced over z regarding x and the language: the lower and upper bounds of z are set as the minimum and maximum costs of any words in $\mathcal{L}_x$.

In summary, `costRegular`$(z, x, \mathcal{L}(\Pi), c)$ dominates its decomposition regular$(x, \mathcal{L}(\Pi)) \wedge$equation$(z, x, c)$. Another decomposition proposed in [Beldiceanu et al., 2005] can easily be generated by introducing intermediary cost variables $\langle z_1, .., z_n\rangle$ and state variables $\langle q_0, .., q_n\rangle$, then posting constraints in extension on each tuple (q_{i-1}, x_i, q_i, z_i) with the Π transition table, and one linear sum $z = z_1 + \cdots + z_n$. In terms of consistency, the two approaches are incomparable (words with costs out of the bounds of z may not be filtered by the decomposition, see examples in [Menana and Demassey, 2009]).

Several API exists for defining the regular language:

- With a deterministic finite automaton (DFA) Π weighted by a cost table c with two dimensions, then $c[i][j]$ is the cost of any transition in Π labeled by j when processing the i-th symbol of a word: it models the cost of assigning variable x_i to value j. The constraint ensures that $z = \sum_i c[i][x_i]$.

- With a DFA Π weighted by a cost table c with three dimensions, then $c[i][j][s]$ is the cost of the transition in Π outgoing from state s and labeled by j when processing the i-th symbol of a word: it models the cost of assigning variable x_i to value j if assignment sequence $\langle x_1, .., x_{i-1}\rangle$ reaches state s when processed by Π. The constraint ensures that $z = \sum_i c[i][x_i][s_i]$ where $\langle s_0, s_1, .., s_n\rangle$ is the sequence of states encountered when recognizing $\langle x_1, .., x_n\rangle$ in Π.
- With a weighted valued multi-graph $G(\Pi)$ and a node s, then $G(\Pi)$ must be a layered graph with $n+1$ layers and s be the unique node in layer 0. Such a graph defines a valued DFA, by setting the arcs as the transitions, the arc values as the transition labels, the arc weights as the transition costs, the nodes as the states, and the nodes in the last layer as the accepting states. Note that this DFA recognizes only words of length n. The constraint ensures that z is the total weight of the path in $G(\Pi)$ produced when recognizing $\langle x_1, .., x_n\rangle$.

Automaton Π is encoded as an object of class `FiniteAutomaton`, whose API contains:

```
FiniteAutomaton();
FiniteAutomaton(String regularExpression);
int addState();
void setInitialState(int state);
void setFinal(int state);
void addTransition(int state1, int state2, int.. labels);
FiniteAutomaton union(FiniteAutomaton a);
FiniteAutomaton intersection(FiniteAutomaton a);
FiniteAutomaton complement();
void minimize();
int getNbStates();
void toDotty(String dotFileName);
```

- **API** :
  - `costRegular(IntegerVariable z, IntegerVariable[] x, FiniteAutomaton pi, int[][] c)`
    based on the `ConstraintType.COSTREGULAR` implementation
  - `costRegular(IntegerVariable z, IntegerVariable[] x, FiniteAutomaton pi, int[][][] c)`
    based on the `ConstraintType.FASTCOSTREGULAR` implementation
  - `costRegular(IntegerVariable z, IntegerVariable[] x, DirectedMultigraph<Node,Arc> g, Node s)`
    based on the `ConstraintType.FASTCOSTREGULAR` implementation
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*
- **references** : [Demassey et al., 2006]: *A* ***`Cost-Regular`*** *based hybrid column generation approach*

Example:
Build the `FiniteAutomaton` manually by adding states and transitions:

```
    // z counts the number of 2 followed by a 0 or a 1 in sequence x
    IntegerVariable[] vars = makeIntVarArray("x", 10, 0, 2, Options.V_ENUM);
    IntegerVariable z = makeIntVar("z", 3, 4, Options.V_BOUND);

    FiniteAutomaton auto = new FiniteAutomaton();
    // states
    int start = auto.addState();
    int end = auto.addState();
    auto.setInitialState(start);
    auto.setFinal(start);
    auto.setFinal(end);
```

```
        // transitions
        auto.addTransition(start, start, 0, 1);
        auto.addTransition(start, end, 2);
        auto.addTransition(end, start, 2);
        auto.addTransition(end, start, 0, 1);
        // costs
        int[][][] costs = new int[vars.length][3][auto.getNbStates()];
        for (int i = 0 ; i < costs.length ; i++) {
            costs[i][0][end] = 1;
            costs[i][1][end] = 1;
        }

        CPModel m = new CPModel();
        m.addConstraint(costRegular(z, vars, auto, costs));
        CPSolver s= new CPSolver();
        s.read(m);
        s.solveAll();
```

Build the FiniteAutomaton from a combination of several regular expressions:

```
        IntegerVariable[] vars = makeIntVarArray("x", 28, 0, 2, Options.V_ENUM);
        IntegerVariable z = makeIntVar("z", 0, 100, Options.V_BOUND);

        // different rules are formulated as patterns that must NOT be matched by x
        List<String> forbiddenRegExps = new ArrayList<String>();
        // do not end with '00' if start with '11'
        forbiddenRegExps.add("11(0|1|2)*00");
        // at most three consecutive 0
        forbiddenRegExps.add("(0|1|2)*0000(0|1|2)*");
        // no pattern '112' at position 5
        forbiddenRegExps.add("(0|1|2){4}112(0|1|2)*");
        // pattern '12' after a 0 or a sequence of 0
        forbiddenRegExps.add("(0|1|2)*02(0|1|2)*");
        forbiddenRegExps.add("(0|1|2)*01(0|1)(0|1|2)*");
        // at most three 2 on consecutive even positions
        forbiddenRegExps.add("(0|1|2)((0|1|2)(0|1|2))*2(0|1|2)2(0|1|2)2(0|1|2)*");

        // a unique automaton is built as the complement language
        // composed of all the forbidden patterns
        FiniteAutomaton auto = new FiniteAutomaton();
        for (String reg : forbiddenRegExps) {
            FiniteAutomaton a = new FiniteAutomaton(reg);
            auto = auto.union(a);
            auto.minimize();
        }
        auto = auto.complement();
        auto.minimize();
        auto.toDotty("myForbiddenRules.dot");
        System.out.println(auto.getNbStates() + "␣states");
        // costs: count the number of 0 and of 1 at odd positions
        int[][] costs = new int[vars.length][3];
        for (int i = 1 ; i < costs.length ; i+=2) {
            costs[i][0] = 1; costs[i][1] = 1;
        }

        CPModel m = new CPModel();
        m.addConstraint(costRegular(z, vars, auto, costs));
        CPSolver s= new CPSolver();
        s.read(m);
        s.minimize(s.getVar(z), true);
        System.out.println(s.solutionToString());
```

7.9 cumulative (constraint)

to be cleaned.

`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:

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

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]`.

For further informations, see the section devoted to this constraint in the Choco Tutorial document.

- **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** :
  - *no option*
  - SettingType.TASK_INTERVAL.getOptionName() for fast task intervals
  - SettingType.SLOW_TASK_INTERVAL.getOptionName() for slow task intervals
  - SettingType.VILIM_CEF_ALGO.getOptionName() for Vilim theta lambda tree + lazy computation of the inner maximization of the edge finding rule of Van hentenrick and Mercier
  - SettingType.VHM_CEF_ALGO_N2K.getOptionName() for Simple $n^2 * k$ algorithm (lazy for R) (CalcEF – Van Hentenrick)
- **favorite domain** : *n/a*
- **references** :
  - [Beldiceanu and Carlsson, 2002] *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, Options.V_BOUND);
    IntegerVariable[] ends = makeIntVarArray("end", n, 0, 6, Options.V_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]});
    }
    TaskVariable[] tasks = Choco.makeTaskVarArray("Task", starts, ends, duration);

    IntegerVariable[] bool = makeIntVarArray("taskIn?", n, 0, 1);
    IntegerVariable obj = makeIntVar("obj", 0, n, Options.V_BOUND, Options.V_OBJECTIVE);
    //post the cumulative
    m.addConstraint(cumulative("cumulative", tasks, height, constant(0), capa,
           SettingType.TASK_INTERVAL.getOptionName()));
    //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());
    }
```

7.10 cumulativeMax (constraint)

Specific case of Cumulative, where the **capacity** is infinite.

- **API** :
  - `cumulativeMax(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable capacity, String... options)`
  - `cumulativeMax(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable capacity, String... options)`
  - `cumulativeMax(TaskVariable[] tasks, int[] heights, int capacity, String... options)`

- **return type** : `Constraint`
- **options** :
  - *no option*
  - SettingType.TASK_INTERVAL.getOptionName() for fast task intervals
  - SettingType.SLOW_TASK_INTERVAL.getOptionName() for slow task intervals
  - SettingType.VILIM_CEF_ALGO.getOptionName() for Vilim theta lambda tree + lazy computation of the inner maximization of the edge finding rule of Van hentenrick and Mercier
  - SettingType.VHM_CEF_ALGO_N2K.getOptionName() for Simple $n^2 * k$ algorithm (lazy for R) (CalcEF – Van Hentenrick)
- **favorite domain** : *n/a*

7.11 cumulativeMin (constraint)

Specific case of Cumulative, where the **consumption** is equal to 0.

- **API** :
  - `cumulativeMin(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, String... options)`
  - `cumulativeMin(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable consumption, String... options)`
  - `cumulativeMin(TaskVariable[] tasks, int[] heights, int consumption, String... options)`
- **return type** : `Constraint`
- **options** :
  - *no option*
  - SettingType.TASK_INTERVAL.getOptionName() for fast task intervals
  - SettingType.SLOW_TASK_INTERVAL.getOptionName() for slow task intervals
  - SettingType.VILIM_CEF_ALGO.getOptionName() for Vilim theta lambda tree + lazy computation of the inner maximization of the edge finding rule of Van hentenrick and Mercier
  - SettingType.VHM_CEF_ALGO_N2K.getOptionName() for Simple $n^2 * k$ algorithm (lazy for R) (CalcEF – Van Hentenrick)
- **favorite domain** : *n/a*

7.12 disjoint (constraint)

`disjoint`$(\langle T^1_1, .., T^1_n \rangle, \langle T^2_1, .., T^2_m \rangle)$ states that each pair of tasks (T^1_i, T^2_j) is in disjunction i.e., the processings of the two tasks do not overlap in time:

$$T^1_i.end \leq T^2_j.start \quad \vee \quad T^2_j.end \leq T^1_i.start, \quad \forall i = 1..n, j = 1..m$$

CHOCO only provides a decomposition with reified precedences because the coloured cumulative is not available.

- **API**: `disjoint(TaskVariable[] t1, TaskVariable[] t2)`
- **return type**: `Constraint[]`
- **favorite domain** : *n/a*.
- **references** :
  - global constraint catalog: `disjoint_tasks`

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
TaskVariable[] tasks1 = Choco.makeTaskVarArray("Task1", 0, 10, new int[]{2,5});
TaskVariable[] tasks2 = Choco.makeTaskVarArray("Task2", 0, 10, new int[]{3,4});
m.addConstraints(disjoint(tasks1, tasks2));
s.read(m);
s.solve();
```

7.13 disjunctive (constraint)

to be cleaned.

`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** :
  - *no option*
  - SettingType.OVERLOAD_CHECKING.getOptionName() overload checking rule (O(n*log(n)), Vilim), also known as task interval
  - SettingType.NF_NL.getOptionName() NotFirst/NotLast rule (O(n*log(n)), Vilim) (recommended)
  - SettingType.DETECTABLE_PRECEDENCE.getOptionName() Detectable Precedence rule (O(n*log(n)), Vilim)
  - SettingType.EDGE_FINDING_D.getOptionName() disjunctive Edge Finding rule (O(n*log(n)), Vilim) (recommended)

  - SettingType.DEFAULT_FILTERING.getOptionName() use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint
  - SettingType.VILIM_FILTERING.getOptionName() use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint
  - SettingType.SINGLE_RULE_FILTERING.getOptionName() use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint. A single filtering rule (debug only).

- **favorite domain** : *n/a*
- **references** :
  global constraint catalog: `disjunctive`

Example: `//TODO: complete`

7.14 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();
```

7.15 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();
```

7.16 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();
```

7.17 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();
```

7.18 domainChanneling (constraint)

`domainChanneling`$(x, \langle b_1, .., b_n \rangle)$ states that boolean b_j is true if and only if x has value j:

$$b_j = 1 \quad \Longleftrightarrow \quad x = j, \qquad \forall j = 1..n$$

It makes the link between a domain variable x and those 0-1 variables b that are associated with each potential value of x: the 0-1 variable b_j associated with the value j taken by x is equal to 1, while the remaining 0-1 variables b_i $(i \neq j)$ are all equal to 0.

- **API** : `domainChanneling(IntegerVariable x, IntegerVariable[] b)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : enumerated for x
- **references** :
  global constraint catalog: `domain_constraint`

Example:

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable x = makeIntVar("var", 0, 10);
        IntegerVariable[] b = makeBooleanVarArray("valueIndicator", 10);
        m.addConstraint(domainChanneling(x, b));
        s.read(m);
        s.solveAll();
```

7.19 element (constraint)

See nth.

7.20 endsAfter (constraint)

`endsAfter`(T, c) states that task T ends after time c:

$$T.end \geq c$$

- **API** :`endsAfter(TaskVariable t, int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*.

Examples: *to complete*

7.21 endsAfterBegin (constraint)

`endsAfterBegin`(T_1, T_2, c) states that task T_1 ends after the start time of T_2 minus c:

$$T_1.end \geq T_2.start - c$$

- **API** :`endsAfterBegin(TaskVariable t1, TaskVariable t2, int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*.

Examples: *to complete*

7.22 endsAfterEnd (constraint)

`endsAfterEnd`(T_1, T_2, c) states that task T_1 ends after the end time of T_2 minus c:

$$T_1.end \geq T_2.end - c$$

- **API** :`endsAfterEnd(TaskVariable t1, TaskVariable t2, int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a.*

Examples: *to complete*

7.23 endsBefore (constraint)

`endsBefore`(T, c) states that task T ends before time c:

$$T.end \leq c$$

- **API** :`endsBefore(final TaskVariable t, final int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a.*

Examples: *to complete*

7.24 endsBeforeBegin (constraint)

`endsBeforeBegin`(T_1, T_2, c) states that task T_1 ends before the start time of T_2 minus c:

$$T_1.end \leq T_2.start - c$$

- **API** :`endsBeforeBegin(TaskVariable t1, TaskVariable t2, int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a.*

Examples: *to complete*

7.25 endsBeforeEnd (constraint)

`endsBeforeEnd`(T_1, T_2, c) states that task T_1 ends before the end time of T_2 minus c:

$$T_1.end \leq T_2.end - c$$

- **API** :`endsBeforeEnd(TaskVariable t1, TaskVariable t2, int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a.*

Examples: *to complete*

7.26 endsBetween (constraint)

`endsBetween`(T, c_1, c_2) states that task T ends between times c_1 and c_2:

$$c_1 \leq T.end \leq c_2$$

- **API** :`endsBetween(TaskVariable t, int min, int max)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a.*

Examples: *to complete*

7.27 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();
```

7.28 eqCard (constraint)

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

$$|s| = z$$

- **API** :
  - `eqCard(SetVariable s, IntegerVariable z)`
  - `eqCard(SetVariable s, int z)`
- **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();
```

7.29 equation (constraint)

`equation`$(z, \langle x_1, .., x_n\rangle, \langle c_1, .., c_n\rangle)$ states that z is the weighted sum of x by c:

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

See also knapsackProblem.

- **API** :
  - equation(int z, IntegerVariable[] x, int[] c)
  - equation(String option, int z, IntegerVariable[] x, int[] c)
  - equation(IntegerVariable z, IntegerVariable[] x, int[] c)
  - equation(String option, IntegerVariable z, IntegerVariable[] x, int[] c)
- **return type** : `Constraint`
- **options** :
  - *no option* or `"cp:ac"`: to enforce GAC using `regular`
  - `"cp:bc"`: to enforce bound consistency using `eq`(z,`scalar`(x, c))
- **favorite domain** : *to complete*
- **global constraint catalog** : scalar_product

Example:

```
CPModel m = new CPModel();
CPSolver s = new CPSolver();
int n = 10;
IntegerVariable[] bvars = makeIntVarArray("b", n, 0, 10, Options.V_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(charge, bvars, coefs);
m.addConstraint(knapsack);
s.read(m);
s.solveAll();
```

7.30 exactly (constraint)

Deprecated, see `occurrence`.

7.31 FALSE (constraint)

FALSE always returns *false*.

7.32 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)
  - `Options.C_EXT_AC3`: to get AC3 algorithm (searching from scratch for supports on all values)
  - `Options.C_EXT_AC2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `Options.C_EXT_AC32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `Options.C_EXT_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<int[]> couples2 = new ArrayList<int[]>();
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(Options.C_EXT_AC32, v1, v2, couples2));
s.read(m);
s.solveAll();
```

7.33 feasTupleAC (constraint)

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

$$\exists \text{ tuple } i \mid \quad \langle x_1,..,x_n\rangle = feasTuples[i]$$

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)
  - `Options.C_EXT_AC32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `Options.C_EXT_AC2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `Options.C_EXT_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<int[]> feasTuple = new ArrayList<int[]>();
feasTuple.add(new int[]{1, 1}); // x*y = 1
feasTuple.add(new int[]{2, 4}); // x*y = 1
m.addConstraint(feasTupleAC(Options.C_EXT_AC2001, feasTuple, v1, v2));
s.read(m);
s.solve();
```

7.34 feasTupleFC (constraint)

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

$$\exists \text{ tuple } i \mid \quad \langle x_1, .., x_n \rangle = feasTuples[i]$$

- **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<int[]> feasTuple = new ArrayList<int[]>();
feasTuple.add(new int[]{1, 1}); // x*y = 1
feasTuple.add(new int[]{2, 4}); // x*y = 1
m.addConstraint(feasTupleFC(feasTuple, v1, v2));
s.read(m);
s.solve();
```

7.35 forbiddenInterval (constraint)

to be detailed

`forbiddenInterval`($\langle t_1, .., t_2 \rangle$) applies additionnal search tree reduction based on time intervals in which no operation can start or end in an optimal solution. *The tasks must all belong to one disjunctive resource and have fixed durations.*

- **API** :
  - `forbiddenInterval(String name, TaskVariable[] tasks)`
  - `forbiddenInterval(TaskVariable[] tasks)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

Example: *to complete*

7.36 geost (constraint)

to be cleaned

`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. The greedy mode should be used without external constraints to have safe results, because it excludes external constraints from its exploration and look for instanciation of variables involved in geost which respect the geost constraint. For further informations, see the section devoted to this constraint in the Choco Tutorial document.

- **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.

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();
```

7.37 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();
```

7.38 geqCard (constraint)

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

$$|s| \geq z$$

- **API** :
  - `geqCard(SetVariable s, IntegerVariable z)`
  - `geqCard(SetVariable s, int z)`
- **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();
```

7.39 globalCardinality (constraint)

`globalCardinality`$(\langle x_1, .., x_n\rangle, \langle l_0, .., l_{M-m}\rangle, \langle u_0, .., u_{M-m}\rangle, m)$ states lower bounds l and upper bounds u on the occurrence numbers of the values in collection x according to offset m:

$$l_{j-m} \;\leq\; |\{i = 1..n \;,\; x_i = j\}| \;\leq\; u_{j-m}, \quad \forall j = m..M$$

`globalCardinality`$(\langle x_1, .., x_n\rangle, \langle z_0, .., z_{M-m}\rangle, o)$ states that z are the occurrence numbers of the values in collection x according to offset m:

$$z_{j-m} = |\{i = 1..n \;,\; x_i = j\}|, \quad \forall j = m..M$$

Note that the length of the bound tables l and u are equal to $M + m - 1$ where m, the offset, is the minimum counted value and M is the maximum counted value.

Several APIs exist:

- *constant bounds on cardinalities* $\langle l_0, .., l_{M-m}\rangle$ and $\langle u_0, .., u_{M-m}\rangle$: use the propagator of [Régin, 1996] or of [Quimper et al., 2003] depending on the set options and the nature of the domain variables.
- *variable cardinalities* $\langle z_0, .., z_{M-m}\rangle$: use the propagator of [Quimper et al., 2003] that:
  - enforces Bound Consistency over x regarding the lower and upper bounds of z,
  - maintains the upper bound of z_j by counting the variables that may be instantiated to j,
  - maintains the lower bound of z_j by counting the variables instantiated to j,

  – enforces $z_0 + \cdots + z_{M-m} = n$

The APIs are duplicated to define options.

- **API** :
  - `globalCardinality(IntegerVariable[] x, int[] low, int[] up, int offset)`
  - `globalCardinality(String options, IntegerVariable[] x, int[] low, int[] up, int offset )`
  - `globalCardinality(IntegerVariable[] x, IntegerVariable[] card, int offset)`
- **return type** : `Constraint`
- **options**:
  - *no option*: if x have *bounded* domains or if the cardinalities are variable z, use the propagator of [Quimper et al., 2003] for BC, otherwise use the propagator of [Régin, 1996];
  - `Options.C_GCC_AC` : for [Régin, 1996] implementation of arc consistency
  - `Options.C_GCC_BC` : for [Quimper et al., 2003] implementation of bound consistency
- **favorite domain** : *enumerated* for arc consistency, *bounded* for bound consistency.
- **references** :
  - [Régin, 1996]: *Generalized arc consistency for global cardinality constraint*,
  - [Quimper et al., 2003]: *An efficient bounds consistency algorithm for the global cardinality constraint*
  - global constraint catalog: global_cardinality

Examples:

- example1:

```
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(Options.C_GCC_BC, globalCardinality(vars, LB2, UB2, 1));
s.read(m);
s.solve();
```

- example2:

```
int n = 5;
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable[] vars = makeIntVarArray("vars", n, 1, n);
IntegerVariable[] cards = makeIntVarArray("cards", n, 0, 1);


m.addConstraint(Options.C_GCC_BC, globalCardinality(vars, cards, 1));
s.read(m);
s.solve();
```

7.40 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();
```

7.41 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(Options.V_BOUND,x ,y, z);
        m.addConstraint(ifOnlyIf(lt(x, y), lt(y, z)));
        s.read(m);
        s.solveAll();
```

7.42 ifThenElse (constraint)

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

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

- **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();
```

7.43 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(Options.V_BOUND,x ,y, z);
        Constraint e1 = implies(leq(x, y), leq(x, z));
        m.addConstraint(e1);
        s.read(m);
        s.solveAll();
```

7.44 increasingNValue (constraint)

`increasingNValue`$(z, \langle x_1, .., x_n\rangle)$ states that $\langle x_1, .., x_n\rangle$ is sorted in increasing order and that z is the number of distinct values occurring in x.

$$z = |\{x_1, \dots, x_n\}| \quad \wedge \quad x_i \leq x_{i+1},\ \forall i = 1..n.$$

- **API** :
  - `increasingNValue(IntegerVariable z, IntegerVariable[] x)`
  - `increasingNValue(String option, IntegerVariable z, IntegerVariable[] x)`
- **return type** : `Constraint`
- **options** :
  - *no option* filter on lower bound and on upper bound
  - `Options.C_INCREASING_NVALUE_ATLEAST` filter on lower bound only
  - `Options.C_INCREASING_NVALUE_ATMOST` filter on upper bound only
  - `Options.C_INCREASING_NVALUE_BOTH` –*default value*– filter on lower bound and on upper bound
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: increasing_nvalue

Example:

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable nval = makeIntVar("nval", 1, 3);
        IntegerVariable[] variables = makeIntVarArray("vars", 6, 1, 4);
        m.addConstraint(increasing_nvalue(Options.C_INCREASING_NVALUE_BOTH, nval, variables));
        s.read(m);
        s.solveAll();
```

7.45 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)
  - `Options.C_EXT_AC3`: to get AC3 algorithm (searching from scratch for supports on all values)
  - `Options.C_EXT_AC2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `Options.C_EXT_AC32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `Options.C_EXT_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(Options.C_EXT_AC32,v1, v2, matrice2));
s.read(m);
s.solveAll();
```

7.46 infeasTupleAC (constraint)

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

$$\forall \text{ tuple } i \mid \quad \langle x_1, .., x_n\rangle \neq infeasTuples[i]$$

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)
  - `Options.C_EXT_AC32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `Options.C_EXT_AC2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `Options.C_EXT_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<int[]> forbiddenTuples = new ArrayList<int[]>();
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, x, y, z));
s.read(m);
s.solveAll();
```

7.47 infeasTupleFC (constraint)

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

$$\forall \text{ tuple } i \mid \quad \langle x_1, .., x_n\rangle \neq infeasTuples[i]$$

- **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<int[]> forbiddenTuples = new ArrayList<int[]>();
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, x, y, z));
s.read(m);
s.solveAll();
```

7.48 intDiv (constraint)

`intDiv`(x, y, z) states that 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();
```

7.49 inverseChanneling (constraint)

`inverseChanneling`$(\langle x_1, .., x_n\rangle, \langle y_1, .., y_m\rangle)$ states that x_i has value j if and only if y_j has value i:

$$x_i = j \iff y_j = i, \qquad \forall i = 1..n, j = 1..m$$

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

Example:

```
int n = 8;
Model m = new CPModel();
IntegerVariable[] queenInCol = new IntegerVariable[n];
IntegerVariable[] queenInRow = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queenInCol[i] = makeIntVar("QC" + i, 1, n);
    queenInRow[i] = makeIntVar("QR" + 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(queenInCol[i], queenInCol[j])); // row
        m.addConstraint(neq(queenInCol[i], plus(queenInCol[j], k))); // diagonal 1
        m.addConstraint(neq(queenInCol[i], minus(queenInCol[j], k))); // diagonal 2
        m.addConstraint(neq(queenInRow[i], queenInRow[j])); // column
        m.addConstraint(neq(queenInRow[i], plus(queenInRow[j], k))); // diagonal 2
        m.addConstraint(neq(queenInRow[i], minus(queenInRow[j], k))); // diagonal 1
    }
}
m.addConstraint(inverseChanneling(queenInCol, queenInRow));
Solver s = new CPSolver();
s.read(m);
s.solveAll();
```

7.50 inverseSet (constraint)

`inverseSet`$(\langle x_1, .., x_n\rangle, \langle s_1, .., s_m\rangle)$ states that x_i has value j if and only if s_j contains value i:

$$x_i = j \quad \Longleftrightarrow \quad i \in s_j, \qquad \forall i = 1..n, j = 1..m$$

- **API** : `inverseSet(IntegerVariable[] x, SetVariable[] s)`
- **return type** : `Constraint`
- **options** : *no options*
- **favorite domain** : enumerated for x
- **references** :
  global constraint catalog: inverse_set

Example:

```
        int i = 4;
        int j = 2;
        Model m = new CPModel();
        IntegerVariable[] iv = makeIntVarArray("iv", i, 0, j);
        SetVariable[] sv = makeSetVarArray("sv", j, 0, i);

        m.addConstraint(inverseSet(iv, sv));
        Solver s = new CPSolver();
        s.read(m);
        s.solveAll();
```

7.51 isIncluded (constraint)

`isIncluded`(s_1, s_2) states that set s_1 is included in set s_2:

$$s_2 \subseteq s_2$$

- **API** : `isIncluded(SetVariable s1, SetVariable s2)`
- **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();
```

7.52 isNotIncluded (constraint)

`isNotIncluded`(s_1, s_2) states that set s_1 is not included in set s_2:

$$s_2 \not\subseteq s_2$$

- **API** : `isNotIncluded(SetVariable s1, SetVariable s2)`
- **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();
```

7.53 knapsackProblem (constraint)

`knapsackProblem`$(z^1, z^2, \langle x_1, .., x_n\rangle, \langle c^1_1, .., c^1_n\rangle, \langle c^2_1, .., c^2_n\rangle)$ states that z^1 (respectively, z^2) is the sum of the x weighted by the costs c^1 (respectively, c^2):

$$\sum_{i=1}^{n} x_i c^1_i = z^1 \quad \wedge \quad \sum_{i=1}^{n} x_i c^2_i = z^2$$

The knaspack problem can be modeled using only this constraint and the objective `maximize(z1)`: x_i is the number of items of type i and each item of type i has a value c^1_i and a weight c^2_i. Based on `costRegular`, this propagator simulates the dynamic programming approach of [Trick, 2003]. It dominates the filtering of the decomposition in two `equation` constraints.

- **API** : `knapsackProblem(IntegerVariable z1, IntegerVariable z2, IntegerVariable[] x, int[] c1 , int[] c2)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*
- **references** : [Trick, 2003]: *A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints*

Example:

```
CPModel m = new CPModel();
IntegerVariable[] items = new IntegerVariable[3];
items[0] = makeIntVar("item_1", 0, 5);
items[1] = makeIntVar("item_2", 0, 7);
items[2] = makeIntVar("item_3", 0, 10);

IntegerVariable sumWeight = makeIntVar("sumWeight", 0, 40, Options.V_BOUND);
IntegerVariable sumValue = makeIntVar("sumValue", 0, 34, Options.V_OBJECTIVE);

int[] weights = new int[]{7, 5, 3};
int[] values = new int[]{6, 4, 2};

Constraint knapsack = Choco.knapsackProblem(items, sumWeight, sumValue, weights, values)
    ;
m.addConstraint(knapsack);

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

7.54 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();
```

7.55 leqCard (constraint)

`leqCard`(s, z) states that the cardinality of set s is less than or equal to z:

$$|s| \leq z$$

- **API** :
  - `leqCard(SetVariable s, IntegerVariable z)`
  - `leqCard(SetVariable s, int z)`

- **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();
```

7.56 lex (constraint)

`lex`$(\langle x_1, .., x_n\rangle, \langle y_1, .., y_n\rangle)$ states a strict lexicographic ordering $x <_{lex} y$:

$$\exists\ j = 1..n \mid \quad x_j < y_j \quad \wedge \quad x_i = y_i\ (\forall\ i < j)$$

- **API** : `lex(IntegerVariable[] x, IntegerVariable[] y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  - [Frisch et al., 2002]: *Global Constraints for Lexicographic Orderings*
  - global constraint catalog: lex_less

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int n = 4;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n];
IntegerVariable[] vs2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
   vs1[i] = makeIntVar("" + i, 0, k);
   vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lex(vs1, vs2));
s.read(m);
s.solve();
```

7.57 lexChain (constraint)

`lexChain`($\langle x^1_1, .., x^1_n\rangle, \ldots, \langle x^p_1, .., x^p_n\rangle$) states a strict lexicographic ordering on a chain of p integer vectors:

$$x^1 <_{lex} x^2 <_{lex} \cdots <_{lex} x^p$$

- **API** : `lexChain(IntegerVariable[]... x)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*
- **references** :
  - [Carlsson and Beldiceanu, 2002] *Arc-Consistency for a chain of Lexicographic Ordering Constraints*
  - global constraint catalog: lex_chain_less

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int n = 4;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n];
IntegerVariable[] vs2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
   vs1[i] = makeIntVar("" + i, 0, k);
   vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lexChain(vs1, vs2));
s.read(m);
s.solve();
```

7.58 lexChainEq (constraint)

`lexChainEq`($\langle x^1_1, .., x^1_n\rangle, \ldots, \langle x^p_1, .., x^p_n\rangle$) states a lexicographic ordering on a chain of p integer vectors:

$$x^1 \leq_{lex} x^2 \leq_{lex} \cdots \leq_{lex} x^p$$

- **API** : `lexChainEq(IntegerVariable[]... x)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *to complete*
- **references** :

- [Carlsson and Beldiceanu, 2002] *Arc-Consistency for a chain of Lexicographic Ordering Constraints*
- global constraint catalog: lex_chain_lesseq

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int n = 4;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n];
IntegerVariable[] vs2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
  vs1[i] = makeIntVar("" + i, 0, k);
  vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lexChainEq(vs1, vs2));
s.read(m);
s.solve();
```

7.59 lexEq (constraint)

`lexEq`$(\langle x_1, .., x_n\rangle, \langle y_1, .., y_n\rangle)$ states a lexicographic ordering $x \leq_{lex} y$:

$$\exists\ j = 1..n \mid \quad x_j \leq y_j \quad \wedge \quad x_i = y_i\ (\forall\ i < j)$$

- **API** : `lexEq(IntegerVariable[] x, IntegerVariable[] y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  - [Frisch et al., 2002]: *Global Constraints for Lexicographic Orderings*
  - global constraint catalog: lex_lesseq

Example: **Example**:

```
Model m = new CPModel();
Solver s = new CPSolver();
int n = 4;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n];
IntegerVariable[] vs2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
   vs1[i] = makeIntVar("" + i, 0, k);
   vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lexEq(vs1, vs2));
s.read(m);
s.solve();
```

7.60 leximin (constraint)

check the specifications of the implemented version.

leximin($\langle x_1,..,x_n\rangle$, $\langle y_1,..,y_n\rangle$) states a strict lexicographic ordering $x' <_{lex} y'$, where x' and y' are the permutations of x and y respectively sorted in increasing order.

$$\texttt{sorting}(\langle x_1,..,x_n\rangle,\langle x'_1,..,x'_n\rangle)\ \wedge\ \texttt{sorting}(\langle y_1,..,y_n\rangle,\langle y'_1,..,y'_n\rangle)\ \wedge\ \texttt{lex}(\langle x'_1,..,x'_n\rangle,\langle y'_1,..,y'_n\rangle)$$

- **API** :
  - `leximin(IntegerVariable[] x, IntegerVariable[] y)`
  - `leximin(int[] x, IntegerVariable[] y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  - [Frisch et al., 2003]: *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();
```

7.61 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();
```

7.62 max (constraint)

7.62.1 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.addVariables(Options.V_BOUND, x, y, z);
m.addConstraint(max(y, z, x));
s.read(m);
s.solve();
```

7.62.2 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*

Example:

```
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();
```

7.63 member (constraint)

member(x, s) states that integer x belongs to 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)`
  - `member(IntegerVariable x, int[] s)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  global constraint catalog: in_set

Examples: 1. using a set variable

```
        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();
```

2. using an array of integers

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable var = makeIntVar("v1", 0, 100, Options.V_BOUND);
        int[] values = new int[]{0, 25, 50, 75, 100};
        m.addConstraint(member(var, values));
        s.read(m);
        s.solve();
```

7.64 min (constraint)

7.64.1 min of a list

`min`(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.addVariables(Options.V_BOUND, x, y, z);
        m.addConstraint(min(y, z, x));
        s.read(m);
        s.solve();
```

7.64.2 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*

Example:

```
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, min), leqCard(set, constant(5)));
s.read(m);
s.solve();
```

7.65 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();
int n = 4;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n];
IntegerVariable[] vs2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
   vs1[i] = makeIntVar("" + i, 0, k);
   vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lex(vs1, vs2));
s.read(m);
s.solve();
```

7.66 multiCostRegular (constraint)

multiCostRegular($\langle z_1,..,z_p\rangle, \langle x_1,..,x_n\rangle, \mathcal{L}(\Pi), \langle c_{i,j,k}\rangle$) states that sequence $\langle x_1,..,x_n\rangle$ is a word belonging to the regular language $\mathcal{L}(\Pi)$ and that each z_k is its cost computed as the sum of the individual symbol weights $c_{i,x_i,k}$:

$$\langle x_1,..,x_n\rangle \in \mathcal{L}(\Pi) \quad \wedge \quad \sum_{i=1}^{n} c_{i,x_i,k} = z_k,\ \forall k = 1..p.$$

multiCostRegular models the conjunction of p costRegular constraints, or the conjunction of a regular constraint with p assignment cost functions. Like them, it is useful for modelling sequencing rules in personnel scheduling and rostering problems. Furthermore it allows to handle together several linear counters and costs on the sequence x. For example, one may count all the value occurrences like with a globalCardinality constraint. Counters can also model assignment costs in optimization problems or violation costs in over-constrained problems. For example, counting the occurrence number of a pattern allows to determine the violation cost of a soft forbidden pattern rule.

The filtering algorithm [Menana and Demassey, 2009] of multiCostRegular does not achieve GAC (as it would be NP-hard) but it dominates the decompositions in regular or costRegular constraints.

The accepting language is specified by a deterministic finite automaton (DFA) Π encoded as an object of class Automaton (see costRegular for a short API). The cost functions are vectors of weights on the transitions of Π. They are encoded as one matrix int c[n][m][p][pi.getNbStates()] such that c[i][j][k][s] is the cost of assigning variable x_i to value j at state s on dimension k. When the transition costs are independent of their initial states, a second API allows to specify a cost matrix int c[n][m][p].

- **API** :
  - multiCostRegular(IntegerVariable[] z, IntegerVariable[] x, FiniteAutomaton pi, int[][][] c)
  - multiCostRegular(IntegerVariable[] z, IntegerVariable[] x, FiniteAutomaton pi, int[][][][] c)
- **return type** : Constraint
- **options** : n/a
- **favorite domain** : *to complete*
- **references** :
  [Menana and Demassey, 2009]: *Sequencing and Counting with the* multicost-regular *Constraint*

Example:

```
import choco.kernel.model.constraints.automaton.FA.Automaton;
```

```
        Model m = new CPModel();

        int nTime = 14; // 2 weeks: 14 days
        int nAct = 3; // 3 shift types:
        int DAY = 0, NIGHT = 1, REST = 2;
        int nCounters = 4; // cost (0), #DAY (1), #NIGHT (2), #WORK (3)

        IntegerVariable[] x = makeIntVarArray("x", nTime, 0, nAct - 1, Options.V_ENUM);

        IntegerVariable[] z = new IntegerVariable[4];
        z[0] = makeIntVar("z", 30, 80, Options.V_BOUND); // 30 <= cost <= 80
        z[1] = makeIntVar("D", 0, 7, Options.V_BOUND); // 0 <= #DAY <= 7
        z[2] = makeIntVar("N", 3, 7, Options.V_BOUND); // 3 <= #NIGHT <= 7
        z[3] = makeIntVar("W", 7, 9, Options.V_BOUND); // 7 <= #WORK <= 9

        FiniteAutomaton auto = new FiniteAutomaton();

        int start = auto.addState();
        auto.setInitialState(start);
        auto.setFinal(start);
        int first = auto.addState();
        auto.addTransition(start, first, DAY); // transition (0,D,1)
        int second = auto.addState();
        auto.addTransition(first, second, DAY, NIGHT); // transitions (1,D,2), (1,N,2)
        auto.addTransition(second, start, REST); // transition (2,R,0)
        auto.addTransition(start, second, NIGHT); // transition (0,N,2)

        int[][][][] c = new int[nTime][nAct][nCounters][auto.getNbStates()];
        for (int i = 0; i < c.length; i++) {
            c[i][DAY][0] = new int[]{3, 1, 0, 1}; // costs of transition (0,D,1)
            c[i][NIGHT][0] = new int[]{8, 0, 1, 1}; // costs of transition (0,N,2)
            c[i][DAY][1] = new int[]{5, 1, 0, 1}; // costs of transition (1,D,2)
            c[i][NIGHT][1] = new int[]{9, 0, 1, 1}; // costs of transition (1,N,2)
            c[i][REST][2] = new int[]{2, 0, 0, 0}; // costs of transition (2,R,0)
        }

        m.addConstraint(multiCostRegular(z, x, auto, c));
        Solver s = new CPSolver();
        s.read(m);
        s.solve();
```

7.67 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();
```

7.68 neqCard (constraint)

`neqCard`(s, z) states that the cardinality of set s is not equal to z:

$$|s| \neq z$$

- **API** :
  - `neqCard(SetVariable s, IntegerVariable z)`
  - `neqCard(SetVariable s, int z)`
- **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(neqCard(set, card));
s.read(m);
s.solve();
```

7.69 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);
m.addConstraint(not(geq(x, 3)));
s.read(m);
s.solve();
```

7.70 notMember (constraint)

`notMember`(x, s) states that integer x does not belong to 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)`
  - `notMember(IntegerVariable x, int[] s)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

- with a set variable

```
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();
```

- with a collection of values

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable var = makeIntVar("v1", 0, 100, Options.V_BOUND);
        int[] values = new int[]{10,20,30,40,50,60,70,80,90};
        m.addConstraint(notMember(var, values));
        s.read(m);
        s.solve();
```

7.71 nth (constraint)

`nth` is the well known *element* constraint. Several APIs are available:

- `nth`$(i, \langle x_0, .., x_n \rangle, y)$ states that $y = x_i$
- `nth`$(i, \langle x_0, .., x_n \rangle, y, o)$ ensures that $y = x_{i-o}$ (o is an *offset* for shifting values)
- `nth`$(i, j, \langle x_{i,j} \rangle, y)$ ensures that $y = x_{i,j}$

- **API** :
  - `nth(IntegerVariable i, int[] x, IntegerVariable y)`
  - `nth(String option, IntegerVariable i, int[] x, IntegerVariable y)`
  - `nth(IntegerVariable i, IntegerVariable[] x, IntegerVariable y)`
  - `nth(IntegerVariable i, int[] x, IntegerVariable y, int offset)`
  - `nth(String option, IntegerVariable i, int[] x, IntegerVariable y, int offset)`
  - `nth(IntegerVariable i, IntegerVariable[] x, IntegerVariable y, int offset)`
  - `nth(String option, IntegerVariable i, IntegerVariable[] x, IntegerVariable y, int offset
)`
  - `nth(IntegerVariable i, IntegerVariable j, int[][] x, IntegerVariable y)`
- **return type** : `Constraint`
- **options** :
  - *no option*
  - `Options.C_NTH_G` for global consistency
- **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();
```

7.72 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 `globalCardinality` with only one value counter, and of `among` with exactly one counted value.

- **API**:
  - `occurrence(IntegerVariable z, IntegerVariable[] x, int v)`
  - `occurrence(int z, IntegerVariable[] x, int v)`
  - `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(y1, new IntegerVariable[]{x1, x2, x3, x4, x5, x6, x7}, 3));
        s.read(m);
        s.solve();
```

7.73 occurrenceMax (constraint)

`occurrenceMax`(v, z, x) states that z is at least equal to the number of elements in x with value v:

$$z \geq |\{i \mid x_i = v\}|$$

See also `occurrence`.

- **API**:
  - `occurrenceMax(IntegerVariable z, IntegerVariable[] x, int v)`
  - `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(y1, new IntegerVariable[]{x1, x2, x3, x4, x5, x6, x7}, 3))
    ;
s.read(m);
s.solve();
```

7.74 occurrenceMin (constraint)

`occurrenceMin`(v, z, x) states that z is at most equal to the number of elements in x with value v:

$$z \leq |\{i \mid x_i = v\}|$$

See also `occurrence`.

- **API**:
  - `occurrenceMin(IntegerVariable z, IntegerVariable[] x, int v)`
  - `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(y1, new IntegerVariable[]{x1, x2, x3, x4, x5, x6, x7}, 3))
            ;
        s.read(m);
        s.solve();
```

7.75 oppositeSign (constraint)

`oppositeSign`(x, y) states that the two arguments have opposite signs:

$$xy < 0$$

Note that 0 has both signs then constraint fails if x or y is equal to 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();
```

7.76 or (constraint)

`or`$(c_1, \ldots, c_n)$ states that at least one constraint in arguments is satisfied:

$$c_1 \vee c_2 \vee \ldots \vee c_n$$

`or`$(b_1, \ldots, b_n)$ states that at least one boolean variable in argument is true:

$$(b_1 = 1) \vee (b_2 = 1) \vee \ldots \vee (b_n = 1)$$

- **API** :
  - `or(Constraint... c)`
  - `or(IntegerVariable... b)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

Examples:

- example1:

```
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();
```

- example2

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable[] vars = makeBooleanVarArray("b", 10);
m.addConstraint(or(vars));
s.read(m);
s.solve();
```

7.77 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 [Shaw, 2004].

- **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.getOptionName(): additional filtering rules *recommended*
  - SettingType.DYNAMIC_LB.getOptionName(): feasibility tests based on dynamic lower bounds for 1D-bin packing
  - SettingType.FILL_BIN.getOptionName(): dominance rule: fill a bin when an item fit into pertfectly equal-sized items and bins must be equivalent
  - SettingType.LAST_BINS_EMPTY.getOptionName(): empty bins are the last ones
- **favorite domain** : *to complete*
- **references** :
  - [Shaw, 2004]: *A constraint for bin packing*
  - global constraint catalog: bin_packing (variant)

Example:
Take a look at *samples.pack* to see advanced use of the constraint.

```
import choco.cp.solver.SettingType;
import choco.cp.solver.search.integer.varselector.StaticVarOrder;
```

```
        Model m = new CPModel();
        m.addConstraint(pack(new int[]{5,3,2,6,8,5},5,10, SettingType.ADDITIONAL_RULES.
            getOptionName()));
        Solver s = new CPSolver();
        s.read(m);
        s.solve();
```

7.78 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:

```
    int k1 = 5;
    Model m = new CPModel();
  IntegerVariable x = makeIntVar("x", 1, 10);
  IntegerVariable y = makeIntVar("y", 1, 10);
  m.addVariables(Options.V_BOUND, x, y);
  IntegerVariable z = makeIntVar("z", 0, 1);
  m.addConstraint(precedenceReified(x,k1,y,z));
    Solver s = new CPSolver();
    s.read(m);
    s.solve();
```

7.79 precedenceimplied (constraint)

To complete

7.80 precedence (constraint)

To complete

7.81 precedencedisjoint (constraint)

To complete

7.82 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.

FiniteAutomaton: is another API for building a DFA (manually, or from a regular expression, or from a `dk.brics.Automaton`) and operating on them (intersection, union, complement) in a more flexible way. Using this API leads to another implementation of the constraint: `FastRegular`. See `costRegular` for a short API of `FiniteAutomaton`.

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 = "(1|2)3{4}5*";`. This expression recognizes any sequences starting by one 1 or one 2, then four consecutive 3 followed by any (possibly empty) sequences of 5.

Warning ! DFA and FiniteAutomaton are both based on the dk.brics library. The construction of these objects is non-deterministic and the order the filtering occur (not the result) may vary at each execution. This may results in different first solutions when branching dynamically using weighted degrees-base heuristics for example.

- **API** :
  - `regular(IntegerVariable[] x, FiniteAutomaton pi)`
  - `regular(IntegerVariable[] x, DFA pi)`
  - `regular(IntegerVariable[] x, List<int[]> feasTuples)`
  - `regular(IntegerVariable[] x, List<int[]> infeasTuples, int[] min, int[] max)`
  - `regular(IntegerVariable[] x, String regexp)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  [Pesant, 2004]: *A regular language membership constraint*

Examples:

- example with `FiniteAutomaton`: see `costRegular`.
- example 1 with DFA:

```java
import choco.kernel.model.constraints.automaton.DFA;
import choco.kernel.model.constraints.automaton.Transition;
import choco.kernel.model.constraints.Constraint;
```

```java
        //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(vars, auto));
        //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(vars, regexp));
        //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());
```

7.83 reifiedAnd (constraint)

To complete

7.84 reifiedConstraint (constraint)

- `reifiedConstraint`(b, C) states that boolean b is true if and only if constraint C holds:

$$(b = 1) \iff C$$

- `reifiedConstraint`(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 negation of constraint of C_1):

$$(b \land C_1) \lor (\neg b \land C_2)$$

- **API** :
  - `reifiedConstraint(IntegerVariable b, Constraint c)`
  - `reifiedConstraint(IntegerVariable b, Constraint c1, Constraint c2)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

The constraint C to reify has to provide its negation $\neg C$ (the negation is needed for propagation). Most basic constraints of Choco provides their negation 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 negation constraint.

The `reifiedConstraint` filter algorithm:

1. if b is instantiated to 1 (resp. to 0), then C (resp. $\neg C$) is propagated
2. otherwise
   (a) if C is entailed, b is set to 1
   (b) else if C is failed, b is set to 0.

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(reifiedConstraint(b, leq(x, y)));
       s.read(m);
       s.solveAll();
```

7.85 reifiedLeftImp (constraint)

To complete

7.86 reifiedNot (constraint)

To complete

7.87 reifiedOr (constraint)

To complete

7.88 reifiedRightImp (constraint)

To complete

7.89 reifiedXnor (constraint)

To complete

7.90 reifiedXor (constraint)

To complete

7.91 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)
  - `Options.C_EXT_AC3`: to get AC3 algorithm (searching from scratch for supports on all values)

  - Options.C_EXT_AC2001: to get AC2001 algorithm (maintaining the current support of each value)
  - Options.C_EXT_AC32: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - Options.C_EXT_AC322: to get AC3 with the used of `BitSet` to know if a support still exists

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

Example:

```
import choco.kernel.solver.constraints.integer.extension.CouplesTest;
import choco.kernel.solver.constraints.integer.extension.TuplesTest;
```

```
   public static class MyEquality extends CouplesTest {

      public boolean checkCouple(int x, int y) {
         return x == y;
      }
   }
```

```
      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(Options.C_EXT_AC32, v1, v2, new MyEquality()));
      m.addConstraint(relationPairAC(Options.C_EXT_AC32, v2, v3, new MyEquality()));
      s.read(m);
      s.solveAll();
```

7.92 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)
  - Options.C_EXT_AC32: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - Options.C_EXT_AC2001: to get AC2001 algorithm (maintaining the current support of each value)
  - Options.C_EXT_AC2008: to get AC2008 algorithm (maintained by STR)

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

Example :

```
  public static 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;
    }
 }
```

```
      Model m = new CPModel();
      IntegerVariable x = makeIntVar("x", 1, 5);
      IntegerVariable y = makeIntVar("y", 1, 5);
    IntegerVariable z = makeIntVar("z", 1, 5);
    m.addConstraint(relationTupleAC(new IntegerVariable[]{x, y, z}, new NotAllEqual()));
      Solver s = new CPSolver();
    s.read(m);
    s.solveAll();
```

7.93 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 :

```
  public static 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;
    }
 }
```

```
      Model m = new CPModel();
      IntegerVariable x = makeIntVar("x", 1, 5);
    IntegerVariable y = makeIntVar("y", 1, 5);
    IntegerVariable z = makeIntVar("z", 1, 5);
    m.addConstraint(relationTupleFC(new IntegerVariable[]{x, y, z}, new NotAllEqual()));
    Solver s = new CPSolver();
      s.read(m);
    s.solveAll();
```

7.94 sameSign (constraint)

`sameSign`(x, y) states that the two arguments have the same sign:

$$xy \geq 0$$

Note that 0 has both signs then constraint holds if x or y is equal to 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());
```

7.95 setDisjoint (constraint)

`setDisjoint`$(s_1, \ldots, s_n)$ states that the arguments are pairwise disjoint:

$$s_i \cap s_j = \emptyset, \quad \forall\, i \neq j$$

- **API** : `setDisjoint(SetVariable[] sv)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

```
      Model m = new CPModel();
      Solver s = new CPSolver();
      SetVariable x = makeSetVar("X", 1, 3);
      SetVariable y = makeSetVar("Y", 1, 3);
```

```
        SetVariable z = makeSetVar("Z", 1, 3);
        Constraint c1 = setDisjoint(x, y, z);
        m.addConstraint(c1);
        s.read(m);
        s.solveAll();
```

7.96 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();
        SetVariable x = makeSetVar("X", 1, 3);
        SetVariable y = makeSetVar("Y", 1, 3);
        SetVariable z = makeSetVar("Z", 2, 3);
        Constraint c1 = setInter(x, y, z);
        m.addConstraint(c1);
        s.read(m);
        s.solveAll();
```

7.97 setUnion (constraint)

`setUnion`(sv, s_{union}) states that the s_{union} set is exactly the union of the sets sv:

$$sv_1 \cup sv_2 \cup \ldots sv_i \cup sv_{i+1} \ldots \cup sv_n = s_{union}$$

- **API** :
  - `setUnion(SetVariable s1, SetVariable s2, SetVariable union)`
  - `setUnion(SetVariable[] sv, SetVariable union)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*

Example:

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        SetVariable x = makeSetVar("X", 1, 3);
        SetVariable y = makeSetVar("Y", 3, 5);
        SetVariable z = makeSetVar("Z", 0, 6);
        Constraint c1 = setUnion(x, y, z);
        m.addConstraint(c1);
        s.read(m);
        s.solveAll();
```

7.98 sorting (constraint)

`sorting`(x, y) holds on the set of variables being either in x or in y, and is satisfied by v if and only if v(y) is the sorted version of v(x) in increasing order.

$$y = x_sorted$$

- **API**: `sorting(IntegerVariable[] x, IntegerVariable[] y)`
- **return type** : `Constraint`
- **options** :*n/a*
- **favorite domain** : *to complete*
- **references** :
  - [Bleuzen-Guernalec and Colmerauer, 1997]: *Narrowing a block of sortings in quadratic time*
  - [Mehlhorn and Thiel, 2000]: *Faster algorithms for bound-consistency of the Sortedness and the Alldifferent constraint*
  - global constraint catalog: count

Example:

```
        CPModel m = new CPModel();
        int n = 3;
        IntegerVariable[] x = makeIntVarArray("x", n, 0, n);
        IntegerVariable[] y = makeIntVarArray("y", n, 0, n);
        m.addConstraint(sorting(x, y));
        m.addConstraint(allDifferent(x));
        CPSolver s = new CPSolver();
        s.read(m);
        s.solveAll();
```

7.99 startsAfter (constraint)

`startsAfter`(T, c) states that the task variable t starts after time c:

$$T.start \geq c$$

- **API** :`startsAfter(final TaskVariable t, final int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*.

Examples: *to complete*

7.100 startsAfterBegin (constraint)

`startsAfterBegin`(T_1, T_2, c) states that task T_1 starts after the start time of T_2 minus c:

$$T_1.start \geq T_2.start - c$$

- **API** :`startsAfterBegin(TaskVariable t1, TaskVariable t2, int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*.

Examples: *to complete*

7.101 startsAfterEnd (constraint)

`startsAfterEnd`(T_1, T_2, c) states that task T_1 starts after the end time of T_2 minus c:

$$T_1.start \geq T_2.end - c$$

- **API** :`startsAfterEnd(TaskVariable t1, TaskVariable t2, int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*.

Examples: *to complete*

7.102 startsBefore (constraint)

`startsBefore`(T, c) states that the task variable T starts before time c:

$$T.start \leq c$$

- **API** :`startsBefore(final TaskVariable t, final int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a.*

Examples: *to complete*

7.103 startsBeforeBegin (constraint)

`startsBeforeBegin`(T_1, T_2, c) states that task T_1 starts before the start time of T_2 minus c:

$$T_1.start \leq T_2.start - c$$

- **API** :`startsBeforeBegin(TaskVariable t1, TaskVariable t2, int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a.*

Examples: *to complete*

7.104 startsBeforeEnd (constraint)

`startsBeforeend`(T_1, T_2, c) states that task T_1 starts before the end time of T_2 minus c:

$$T_1.start \leq T_2.end - c$$

- **API** :`startsBeforeEnd(TaskVariable t1, TaskVariable t2, int c)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a.*

Examples: *to complete*

7.105 startsBetween (constraint)

`startsBetween`(T, c_1, c_2) states that task T starts between times c_1 and c_2:

$$c_1 \leq T.start \leq c_2$$

- **API** :`startsBetween(TaskVariable t, int min, int max)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a.*

Examples: *to complete*

7.106 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** :
  - [Pesant, 2004]: *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();
```

7.107 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));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

7.108 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** :
  - [Beldiceanu et al., 2008]: *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:

```
import choco.kernel.model.variables.tree.TreeParametersObject;
```

```
        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, s.getVar(parameters.getSuccVars())));
        s.solveAll();
```

7.109 TRUE (constraint)

TRUE always returns *true*.

7.110 xnor (constraint)

`xnor`(b_1, b_2) states that two booleans are either both true, or both false:

$$(b_1 = 1) \quad \Longleftrightarrow \quad (b_2 = 1)$$

- **API** : `xnor(IntegerVariable b1, IntegerVariable b2)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

Examples:

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

7.111 xor (constraint)

`xor`(b_1, b_2) states that two booleans are the one true and the other one false:

$$(b_1 = 1) \iff (b_2 = 0)$$

- **API** : `xor(IntegerVariable b_1, IntegerVariable b_2)`
- **return type** : `Constraint`
- **options** : *n/a*
- **favorite domain** : *n/a*

Examples:

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

Chapter 8

Options (Model)

This section lists and details the options that can be declared on variables or constraints within a Choco Model.

8.1 Options and settings

The variables and some constraints allow the declaration of options. Options and settings are defined in the classes `Options` and `SettingType`. Default options are specified. Most of the time, options parameters are *varargs*.

8.1.1 Options for variables:

- `Options.NO_OPTION`, `""`, or *empty argument*
- `Options.V_BOUND` or `"cp:bound"`
  - **goal** : force the solver to create bounded domain 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.
  - **scope** :
    * IntegerVariable
    * SetVariable
    * TaskVariable
- `Options.V_ENUM` or `"cp:enum"`
  - **goal** : force the solver to create enumerated domain variable *(default option)*. 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.
  - **scope** :
    * IntegerVariable
    * SetVariable
    * TaskVariable
- `Options.V_BTREE` or `"cp:btree"`
  - **goal** : force the solver to create binary tree domain variable. *Under development*
  - **scope** : IntegerVariable
- `Options.V_BLIST` or `"cp:blist"`

- **goal** : force the solver to create bipartite list domain variable. It is a domain where unavailable values are placed in the left part of the list, the other one on the right one.
- **scope** : IntegerVariable

- `Options.V_LINK` or `"cp:link"`
  - **goal** : force the solver to create linked list domain 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.
  - **scope** : IntegerVariable
- `Options.V_MAKESPAN` or `"cp:makespan"`
  - **goal** : declare the current variable as makespan
  - **scope** : IntegerVariable
- `Options.V_NO_DECISION` or `"cp:no_decision"`
  - **goal** : force variable to be removed from the pool of decisionnal variables of the default search strategy
  - **scope** :
    - IntegerVariable
    - SetVariable
    - RealVariable
    - TaskVariable
- `Options.V_OBJECTIVE` or `"cp:objective"`
  - **goal** : declare the objective variable
  - **scope** :
    - IntegerVariable
    - SetVariable
    - RealVariable

8.1.2 Options for expressions:

- `Options.NO_OPTION`, `""`, or *empty argument*
- `Options.E_DECOMP` or `"cp:decomp"`
  - **goal** : force decomposition of the scoped expression.
  - **scope** : IntegerExpressionVariable

8.1.3 Options and settings for constraints:

- `Options.NO_OPTION`, `""`, or *empty argument*
- `Options.C_EXT_AC3` or `"cp:ac3"`
  - **goal** : to get AC3 algorithm (searching from scratch for supports on all values)
  - **scope** :
    - feasPairAC(String, IntegerVariable, IntegerVariable, List)
    - infeasPairAC(String, IntegerVariable, IntegerVariable, List)

    * infeasPairAC(String, IntegerVariable, IntegerVariable, boolean[][])
    * relationPairAC(String, IntegerVariable, IntegerVariable, BinRelation)

- `Options.C_EXT_AC32` or `"cp:ac32"`
  - **goal** : to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - **scope** :
    * feasPairAC(String, IntegerVariable, IntegerVariable, List)
    * feasTupleAC(String, List, IntegerVariable[])
    * infeasPairAC(String, IntegerVariable, IntegerVariable, List)
    * infeasPairAC(String, IntegerVariable, IntegerVariable, boolean[][])
    * relationPairAC(String, IntegerVariable, IntegerVariable, BinRelation)
    * relationTupleAC(String, IntegerVariable[], LargeRelation)
- `Options.C_EXT_AC322` or `"cp:ac322"`
  - **goal** : to get AC3 with the used of `BitSet` to know if a support still exists
  - **scope** :
    * feasPairAC(String, IntegerVariable, IntegerVariable, List)
    * infeasPairAC(String, IntegerVariable, IntegerVariable, List)
    * infeasPairAC(String, IntegerVariable, IntegerVariable, boolean[][])
    * relationPairAC(String, IntegerVariable, IntegerVariable, BinRelation)
- `Options.C_EXT_AC2001` or `"cp:ac2001"`
  - **goal** : to get AC2001 algorithm (maintaining the current support of each value)
  - **scope** :
    * feasPairAC(String, IntegerVariable, IntegerVariable, List)
    * feasTupleAC(String, List, IntegerVariable[])
    * infeasPairAC(String, IntegerVariable, IntegerVariable, List)
    * infeasPairAC(String, IntegerVariable, IntegerVariable, boolean[][])
    * relationPairAC(String, IntegerVariable, IntegerVariable, BinRelation)
    * relationTupleAC(String, IntegerVariable[], LargeRelation)
- `Options.C_EXT_AC2008` or `"cp:ac2008"`
  - **goal** : to get AC2008 algorithm (maintained by STR)
  - **scope** :
    * feasTupleAC(String, List, IntegerVariable[])
    * infeasTupleAC(String, List, IntegerVariable[])
    * relationTupleAC(String, IntegerVariable[], LargeRelation)
- `Options.C_EXT_FC` or `"cp:fc"`
  - **goal** : set filter policy to forward checking
  - **scope** :
    * feasTupleAC(String, List, IntegerVariable[])
    * infeasTupleAC(String, List, IntegerVariable[])
    * relationTupleAC(String, IntegerVariable[], LargeRelation)
- `Options.C_ALLDIFFERENT_AC` or `"cp:ac"`
  - **goal** : for Regin implementation

  - **scope** : allDifferent(String, IntegerVariable[])

- `Options.C_ALLDIFFERENT_BC` or `"cp:bc"`
  - **goal** : for bound all different using the propagator of A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. IJCAI-2003
  - **scope** : allDifferent(String, IntegerVariable[])
- `Options.C_ALLDIFFERENT_CLIQUE` or `"cp:clique"`
  - **goal** : propagate on the clique of differences
  - **scope** : allDifferent(String, IntegerVariable[])
- `Options.C_GCC_AC` or `"cp:ac"`
  - **goal** : for Regin implementation
  - **scope** : globalCardinality(String, IntegerVariable[], int[], int[], int)
- `Options.C_GCC_BC` or `"cp:bc"`
  - **goal** : for Quimper implementation
  - **scope** : globalCardinality(String, IntegerVariable[], int[], int[], int)
- `Options.C_INCREASING_NVALUE_ATLEAST` or `"cp:atleast"`
  - **goal** : set filtering policy to filter on lower bound only
  - **scope** : increasing_nvalue(String, IntegerVariable, IntegerVariable[])
- `Options.C_INCREASING_NVALUE_ATMOST` or `"cp:atmost"`
  - **goal** : set filtering policy to filter on upper bound only
  - **scope** : increasing_nvalue(String, IntegerVariable, IntegerVariable[])
- `Options.C_INCREASING_NVALUE_BOTH` or `"cp:both"`
  - **goal** : set filtering policy to filter on lower and upper bound only
  - **scope** : increasing_nvalue(String, IntegerVariable, IntegerVariable[])
- `Options.C_NTH_G` or `"cp:G"`
  - **goal** : global consistency
  - **scope** :
    * nth(String options, IntegerVariable index, int[] values, IntegerVariable val)
    * nth(String option, IntegerVariable index, IntegerVariable[] varArray, IntegerVariable val)
    * nth(String options, IntegerVariable index, IntegerVariable[] varArray, IntegerVariable val, int offset)
- `Options.C_CLAUSES_ENTAIL` or `"cp:entail"`
  - **goal** : ensures quick entailment tests
  - **scope** : clause(IntegerVariable[],IntegerVariable[])
- `Options.C_POST_PONED` or `"cp:postponed"`
  - **goal** : postponed a constraint
  - **scope** : `Constraint`
- `SettingType.ADDITIONAL_RULES` or `"cp:pack:ar"`

  - **goal** : more filtering rules (recommended)
  - **scope** :
    * pack(int[] sizes, int nbBins, int capacity, String... options)
    * pack(PackModeler modeler,String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes, String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes,IntegerVariable nbNonEmpty, String... options)

- `SettingType.DYNAMIC_LB` or `"cp:pack:dlb"`
  - **goal** : feasibility test based on a dynamic lower bound
  - **scope** :
    * pack(int[] sizes, int nbBins, int capacity, String... options)
    * pack(PackModeler modeler,String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes, String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes,IntegerVariable nbNonEmpty, String... options)

- `SettingType.FILL_BIN` or `"cp:pack:fill"`
  - **goal** : dominance rule: fill a bin when an item fit into pertfectly equal-sized items and bins must be equivalent
  - **scope** :
    * pack(int[] sizes, int nbBins, int capacity, String... options)
    * pack(PackModeler modeler,String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes, String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes,IntegerVariable nbNonEmpty, String... options)

- `SettingType.LAST_BINS_EMPTY` or `"cp:pack:lbe"`
  - **goal** : empty bins are the last ones
  - **scope** :
    * pack(int[] sizes, int nbBins, int capacity, String... options)
    * pack(PackModeler modeler,String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes, String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes,IntegerVariable nbNonEmpty, String... options)

- `SettingType.TASK_INTERVAL` or `"cp:cumul:ti"`
  - **goal** : for fast task intervals
  - **scope** :
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, IntegerVariable uppBound, String... options)
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, String... options)
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable consumption, IntegerVariable capacity, String... options)

- `SettingType.SLOW_TASK_INTERVAL` or `"cp:cumul:sti"`
  - **goal** : for slow task intervals
  - **scope** :
    - cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, IntegerVariable uppBound, String... options)
    - cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, String... options)
    - cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable consumption, IntegerVariable capacity, String... options)
- `SettingType.VILIM_CEF_ALGO` or `"cp:cumul:cef"`
  - **goal** : for Vilim theta lambda tree + lazy computation of the inner maximization of the edge finding rule of Van hentenrick and Mercier
  - **scope** :
    - cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, IntegerVariable uppBound, String... options)
    - cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, String... options)
    - cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable consumption, IntegerVariable capacity, String... options)
- `SettingType.VHM_CEF_ALGO_N2K` or `"cp:cumul:scef"`
  - **goal** : for Simple $n^2 * k$ algorithm (lazy for R) (CalcEF – Van Hentenrick)
  - **scope** :
    - cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, IntegerVariable uppBound, String... options)
    - cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, String... options)
    - cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable consumption, IntegerVariable capacity, String... options)
- `SettingType.OVERLOAD_CHECKING` or `"cp:unary:oc"`
  - **goal** : overload checking rule (O(n*log(n)), Vilim), also known as task interval
  - **scope** :
    - disjunctive(TaskVariable[] tasks, String... options)
    - disjunctive(String name, TaskVariable[] tasks, String... options)
    - disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    - disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    - disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)
- `SettingType.NF_NL` or `"cp:unary:nfnl"`
  - **goal** : NotFirst/NotLast rule (O(n*log(n)), Vilim) (recommended).
  - **scope** :
    - disjunctive(TaskVariable[] tasks, String... options)
    - disjunctive(String name, TaskVariable[] tasks, String... options)

    * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)

- `SettingType.DETECTABLE_PRECEDENCE` or `"cp:unary:dp"`
  - **goal** : Detectable Precedence rule (O(n*log(n)), Vilim).
  - **scope** :
    * disjunctive(TaskVariable[] tasks, String... options)
    * disjunctive(String name, TaskVariable[] tasks, String... options)
    * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)
- `SettingType.EDGE_FINDING_D` or `"cp:unary:ef"`
  - **goal** : disjunctive Edge Finding rule (O(n*log(n)), Vilim) (recommended).
  - **scope** :
    * disjunctive(TaskVariable[] tasks, String... options)
    * disjunctive(String name, TaskVariable[] tasks, String... options)
    * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)
- `SettingType.DEFAULT_FILTERING` or `"cp:unary:df"`
  - **goal** : use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint.
  - **scope** :
    * disjunctive(TaskVariable[] tasks, String... options)
    * disjunctive(String name, TaskVariable[] tasks, String... options)
    * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)
- `SettingType.VILIM_FILTERING` or `"cp:unary:vf"`
  - **goal** : use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint.
  - **scope** :
    * disjunctive(TaskVariable[] tasks, String... options)
    * disjunctive(String name, TaskVariable[] tasks, String... options)
    * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)
- `SettingType.SINGLE_RULE_FILTERING` or `"cp:unary:srf"`
  - **goal** : use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint. A single filtering rule (debug only).

- **scope** :
    - disjunctive(TaskVariable[] tasks, String... options)
    - disjunctive(String name, TaskVariable[] tasks, String... options)
    - disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    - disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    - disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)

8.1.4 Options for solvers:

- `Options.NO_OPTION`, `""`, or *empty argument*
- `Options.S_MULTIPLE_READINGS` or `"cp:multiple_readings"`
    - **goal** : Allow a solver to read a model more than one time. *In that case, the redundant constraints for scheduling must be posted explicitly.*
    - **scope** : `CPSolver`

Chapter 9

Branching strategies (Solver)

This section lists and details the branching strategies currently available in Choco.

9.1 AssignInterval (Branching strategy)

`AssignInterval` is a **binary branching** assigning two distinct intervals to a real variable. Following the *interval bisection rule*, the interval representing the domain of the selected variable is split into two parts at its midpoint. In the first branch, the variable upper bound is set to the midpoint; in the second branch, the variable lower bound is set to the midpoint.

$$B_1 : x \in [\underline{x}, m], \quad B_2 : x \in [m, \overline{x}], \qquad \text{with } m = \frac{\underline{x} + \overline{x}}{2}$$

- **Constructor** :`AssignInterval(VarSelector<RealVar> varSel, ValIterator<RealVar> valIt)`
- **type of variable** : real
- **references** : *n/a*

Example:

9.2 AssignOrForbidIntVarVal (Branching strategy)

`AssignOrForbidIntVarVal` is a **binary branching** assigning a value to an integer variable. In the first branch, the selected value is assigned to the selected variable; in the second branch, the value is removed from the variable domain.

$$B_1 : x = v, \quad B_2 : x \neq v$$

- **Constructor** :`AssignOrForbidIntVarVal(VarSelector<IntDomainVar> varSel, ValSelector<IntDomainVar> valSel)`
- **type of variable** : integer
- **references** : *n/a*

Example:

9.3 AssignOrForbidIntVarValPair (Branching strategy)

`AssignOrForbidIntVarValPair` is a **binary branching** assigning a value to an integer variable. In the first branch, the selected value is assigned to the selected variable; in the second branch, the value is removed from the variable domain. It requires a `VarValPairSelector` selecting both variable and value at the same time.

$$B_1 : x = v, \quad B_2 : x \neq v$$

- **Constructor** :`AssignOrForbidIntVarValPair(VarValPairSelector varValSel)`
- **type of variable** : integer
- **references** : *n/a*
- **see also** : `AssignOrForbidIntVarVal` based on distinct variable/value selectors

Example:

9.4 AssignSetVar (Branching strategy)

`AssignSetVar` is a **n-ary branching** assigning distinct values to a set variable. The selected variable is successively assigned, in each branch, to a next selected value.

$$B_1 : x = v_1, \quad B_2 : x = v_2, \quad \dots, \quad B_m : x = v_m$$

- **Constructor** :`AssignSetVar(VarSelector<SetVar> varSel, ValSelector<SetVar> valSel)`
- **type of variable** : set
- **references** : *n/a*

Example:

9.5 AssignVar (Branching strategy)

`AssignVar` is an **n-ary branching** assigning distinct values to an integer variable. The selected variable is successively assigned, in each branch, to a next selected value.

$$B_1 : x = v_1, \quad B_2 : x = v_2, \quad \dots, \quad B_m : x = v_m$$

- **Constructor** :
  - `AssignVar(VarSelector<IntDomainVar> varSel, ValSelector<IntDomainVar> valSel)`
  - `AssignVar(VarSelector<IntDomainVar> varSel, ValIterator<IntDomainVar> valIt)`
- **type of variable** : integer
- **references** : *n/a*

Example:

9.6 DomOverWDegBranchingNew (Branching strategy)

`DomOverWDegBranchingNew` is a **n-ary branching** assigning distinct values to an integer variable. It maintains (incrementally or dynamically) on each constraint, the count of the failures caused by the constraint from the beginning of the search. To each variable are then associated, at any time, three values: *dom* the current domain size, *deg* the current number of uninstantiated constraints involving the variable, and w the sum of the counters associated with these *deg* constraints. The strategy selects the variable with the smallest ratio $r_i = dom/w * deg$. Ties are randomly broken when `seed != null`. The variable is then successively assigned, in each branch, to a next selected value.

$$B_1 : x = v_1, \quad B_2 : x = v_2, \quad \ldots, \quad B_m : x = v_m, \qquad \text{with } x = \arg\min\left(\frac{dom_x}{w_x * deg_x}\right)$$

- **Constructor** :`DomOverWDegBranchingNew(Solver s, IntDomainVar[] vars, ValIterator valIt, Number seed)`
- **type of variable** : integer
- **references** : [Boussemart et al., 2004]: *Boosting systematic search by weighting constraints.*

Example:

9.7 DomOverWDegBinBranchingNew (Branching strategy)

`DomOverWDegBranchingNew` is a **binary branching** assigning distinct values to an integer variable. It maintains (incrementally or dynamically) on each constraint, the count of the failures caused by the constraint from the beginning of the search. To each variable are then associated, at any time, three values: *dom* the current domain size, *deg* the current number of uninstantiated constraints involving the variable, and w the sum of the counters associated with these *deg* constraints. The strategy selects the variable with the smallest ratio $r_i = dom/w * deg$. Ties are randomly broken when `seed != null`. The variable is then assigned, in the first branch, to the selected value; in the second branch, the value is removed from the variable domain.

$$B_1 : x = v, \quad B_2 : x \neq v, \qquad \text{with } x = \arg\min\left(\frac{dom_x}{w_x * deg_x}\right)$$

- **Constructor**: `DomOverWDegBinBranchingNew(Solver s, IntDomainVar[] vars, ValSelector valSel, Number seed)`
- **type of variable** : integer
- **references** : [Boussemart et al., 2004]: *Boosting systematic search by weighting constraints.*

Example:

9.8 ImpactBasedBranching (Branching strategy)

`ImpactBasedBranching` is a **n-ary branching** assigning all distinct values to an integer variable. The impact of a branching decision measures the reduction of the search space induced when the decision was posted and propagated since the beginning of the search. Here, branching decisions are variable-value assignments. The strategy selects an integer variable maximizing the total impact minus the domain size. The variable is then successively assigned, in each branch, to its possible values selected by decreasing order of their impacts. *Restarting search can dramatically improve performance.*

$$B_1 : x = v_1, \quad B_2 : x = v_2, \quad \ldots, \quad B_m : x = v_m,$$

with $x = \arg\max \sum_{v \in D(x)} (impact(x, v) - 1)$ and $impact(x, v_i) > impact(x, v_j), \forall j < i$.

- **Constructor** :
  - `ImpactBasedBranching(Solver s, IntDomainVar[] vars)`
  - `ImpactBasedBranching(Solver s, IntDomainVar[] vars, AbstractImpactStrategy ibs)`
- **type of variable** : integer
- **references** : [Refalo, 2004]: *Impact-Based Search Strategies for Constraint Programming.*

Example:

9.9 PackDynRemovals (Branching strategy)

`PackDynRemovals` is a **n-ary branching** for packing problems with identical bin capacities and without pre-assignments, placing an item in distinct bins. It is a specialization of AssignVar for breaking symmetries: the selected item x is successively placed, in each branch, in a next selected bin. At every backtrack, once a bin has been tried, all other bins having exactly the same residual space *rcap* become unselectable.

$$B_1 : bin(x) = v_1, \quad B_2 : bin(x) = v_2, \quad \ldots, \quad B_n : bin(x) = v_n, \qquad \text{with } rcap(v_j) \neq rcap(v_i)\ \forall i < j$$

- **Constructor**:`PackDynRemovals(VarSelector varSel, ValSelector valSel, PackSConstraint pack)`
- **type of variable** : integer *(bin-to-item assignment variable)*
- **references** : *n/a*

Example:

9.10 SetTimes (Branching strategy)

`SetTimes` is a **n-ary branching** for scheduling problems with makespan minimization objective, fixing distinct task variables to their earliest starting time. At each node, a set of available tasks is considered then, in each branch, one task variable is selected and its starting time is set to its smallest value. The task variables are selected in order according to a `TaskVarSelector`, or in the decreasing order defined by a `Comparator`, ties being randomly broken when boolean `rand` is set. *The search is not complete: do not use within a* `solveAll`.

$$B_1 : T_1.start = \underline{T_1.start}, \quad B_2 : T_2.start = \underline{T_2.start}, \quad \dots, \quad B_n : T_n.start = \underline{T_n.start}$$

- **Constructor** :
  - `SetTimes(Solver solver, List<TaskVar> tasks, TaskVarSelector varSel)`
  - `SetTimes(Solver solver, List<TaskVar> tasks, Comparator<ITask> comp, boolean rand)`
- **type of variable** : task
- **references** : *n/a*

Example:

9.11 TaskOverWDegBinBranching (Branching strategy)

`TaskOverWDegBinBranching` is a **binary branching** for scheduling problems, fixing the relative order between two task variables constrained by a precedence relation. Like in `DomOverWDegBranchingNew`, the pair of task variables is selected according to the ratio of the size of the variable domains and the failure weight of the constraints involving these variables. In the first branch, the selected pair of task variables is ordered according to the value returned by the `OrderingValSelector` (1 if T_1 precedes T_2 or 0 if T_2 precedes T_1); in the second branch, the opposite relative order is enforced.

$$B_1 : \texttt{precedence}(T_1, T_2, order), \quad B_2 : \texttt{precedence}(T_1, T_2, 1 - order)$$

- **Constructor** :`TaskOverWDegBinBranching(Solver s, IPrecedenceRatio[] varRatios, OrderingValSelector valOrd, Number seed)`
- **type of variable** : task *(actually: precedence boolean indicator)*
- **references** : [Boussemart et al., 2004]: *Boosting systematic search by weighting constraints.*

Example:

Chapter 10

Variable selectors (Solver)

This section lists and details the variable selectors currently available in Choco.

10.1 CompositeIntVarSelector (Variable selector)

`CompositeIntVarSelector`(h_1, h_2) selects a constraint according to heuristic h_1, then selects an integer variable involved in the constraint according to heuristic h_2:

$$h_2(support(h_1))$$

- **Constructor** :`CompositeIntVarSelector(ConstraintSelector h1, HeuristicIntVarSelector h2)`
  - `ConstraintSelector` is an `interface`. No implementation provided.
  - `HeuristicIntVarSelector` is an `abstract class` implemented by: MinDomain, MaxDomain, MaxRegret, MaxValueDomain, MinValueDomain, MostConstrained. list to complete
- **type of variable** : integer
- **references** : *n/a*

Example:

10.2 CyclicRealVarSelector (Variable selector)

`CyclicRealVarSelector`(x) selects the real variables in the order they appear in array x, in a cyclic way until they are all instantiated, i.e. with interval domain under the desired precision (**static**).

- **Constructor** :
  - `CyclicRealVarSelector(Solver s)`
  - `CyclicRealVarSelector(Solver s, RealVar[] vars)`
- **type of variable** : real
- **references** : *n/a*

Example:

10.3 LexIntVarSelector (Variable selector)

`LexIntVarSelector`(h_1, h_2) selects the integer variable according to the first heuristic h_1, ties being broken by the second heuristic h_2:

$$\begin{cases} h_1(x) & \text{if } |h_1(x)| = 1, \\ h_2(h_1(x)) & \text{otherwise.} \end{cases}$$

- **Constructor**: `LexIntVarSelector(TiedIntVarSelector h1, HeuristicIntVarSelector h2)`
  - `TiedIntVarSelector` is an `interface` implemented by : MinDomain, MaxDomain, MaxRegret, MaxValueDomain, MinValueDomain, MostConstrained, RandomIntVarSelector.
  - `HeuristicIntVarSelector` is an `abstract class` implemented by: MinDomain, MaxDomain, MaxRegret, MaxValueDomain, MinValueDomain, MostConstrained, RandomIntVarSelector.
- **type of variable** : integer
- **references** : *n/a*

Example:

10.4 MaxDomain (Variable selector)

`MaxDomain`(x) selects the integer variable with the largest domain (**dynamic**):

$$\max |D(x)|$$

- **Constructor** :
  - `MaxDomain(Solver s)`
  - `MaxDomain(Solver s, IntDomainVar[] vars)`
- **type of variable** : integer
- **references** : *n/a*

Example:

10.5 MaxDomSet (Variable selector)

`MaxDomSet`(x) selects the set variable with the largest open domain (**dynamic**):

$$\max |Env(x) \setminus Ker(x)|$$

- **Constructor** :

  - `MaxDomSet(Solver s)`
  - `MaxDomSet(Solver s, SetVar[] vars)`
- **type of variable** : set
- **references** : *n/a*

Example:

10.6 MaxRegret (Variable selector)

`MaxRegret`(x) selects the integer variable with the largest difference between the two smallest values in its domain (**dynamic**):

$$\max(\underline{x}_2 - \underline{x}), \qquad \text{with } \underline{x}_2 = \min(D(x) \setminus \underline{x})$$

- **Constructor** :
  - `MaxRegret(Solver s)`
  - `MaxRegret(Solver s, IntDomainVar[] vars)`
- **type of variable** : integer
- **references** : *n/a*

Example:

10.7 MaxRegretSet (Variable selector)

`MaxRegretSet`(x) selects the set variable with the largest difference between the two smallest values in its envelope (**dynamic**):

$$\max(\underline{x}_2 - \underline{x}), \qquad \text{with } \underline{x} = \min(Env(x)), \underline{x}_2 = \min(Env(x) \setminus \underline{x})$$

- **Constructor** :
  - `MaxRegretSet(Solver s)`
  - `MaxRegretSet(Solver s, SetVar[] vars)`
- **type of variable** : set
- **references** : *n/a*

Example:

10.8 MaxValueDomain (Variable selector)

`MaxValueDomain`(x) selects the integer variable with the largest value in its domain (**dynamic**):

$$\max(\bar{x})$$

- **Constructor** :
  - `MaxValueDomain(Solver s)`
  - `MaxValueDomain(Solver s, IntDomainVar[] vars)`
- **type of variable** : integer
- **references** : *n/a*

Example:

10.9 MaxValueDomSet (Variable selector)

`MaxValueDomSet`(x) selects the set variable with the largest value in its envelope (**dynamic**):

$$\max(\bar{x}), \qquad \text{with } \bar{x} = \max(Env(x))$$

- **Constructor** :
  - `MaxValueDomSet(Solver s)`
  - `MaxValueDomSet(Solver s, SetVar[] vars)`
- **type of variable** : set
- **references** : *n/a*

Example:

10.10 MinDomain (Variable selector)

`MinDomain`(x) selects the integer variable with the smallest domain (**dynamic**):

$$\min |D(x)|$$

- **Constructor** :
  - `MinDomain(Solver s)`
  - `MinDomain(Solver s, IntDomainVar[] vars)`
- **type of variable** : integer
- **references** : *n/a*

Example:

10.11 MinDomSet (Variable selector)

`MinDomSet`(x) selects the set variable with the smallest open domain (**dynamic**):

$$\min |Env(x) \setminus Ker(x)|$$

- **Constructor** :
  - `MinDomSet(Solver s)`
  - `MinDomSet(Solver s, SetVar[] vars)`
- **type of variable** : set
- **references** : *n/a*

Example:

10.12 MinValueDomain (Variable selector)

`MinValueDomain`(x) selects the integer variable with the smallest value in its domain (**dynamic**):

$$\min(\underline{x})$$

- **Constructor** :
  - `MinValueDomain(Solver s)`
  - `MinValueDomain(Solver s, IntDomainVar[] vars)`
- **type of variable** : integer
- **references** : *n/a*

Example:

10.13 MinValueDomSet (Variable selector)

`MinValueDomSet`(x) selects the set variable with the smallest value in its envelope (**dynamic**):

$$\min(\underline{x}), \qquad \text{with } \underline{x} = \min(Env(x))$$

- **Constructor** :
  - `MinValueDomSet(Solver s)`
  - `MinValueDomSet(Solver s, SetVar[] vars)`
- **type of variable** : set
- **references** : *n/a*

Example:

10.14 MostConstrained (Variable selector)

`MostConstrained`(x) selects the integer variable involved in the largest number of constraints initially present in the solver (**static**):

$$\max(initDeg(x))$$

- **Constructor** :
  - `MostConstrained(Solver s)`
  - `MostConstrained(Solver s, IntDomainVar[] vars)`
- **type of variable** : integer
- **references** : *n/a*

Example:

10.15 MostConstrainedSet (Variable selector)

`MostConstrainedSet`(x) selects the set variable involved in the largest number of constraints initially present in the solver (**static**):

$$\max(initDeg(x))$$

- **Constructor** :
  - `MostConstrainedSet(Solver s)`
  - `MostConstrainedSet(Solver s, SetVar[] vars)`
- **type of variable** : set
- **references** : *n/a*

Example:

10.16 RandomIntVarSelector (Variable selector)

`RandomIntVarSelector`(x) selects an integer variable randomly (**dynamic**). The random `seed` can be fixed.

- **Constructor** :
  - `RandomIntVarSelector(Solver s)`
  - `RandomIntVarSelector(Solver s, long seed)`
  - `RandomIntVarSelector(Solver s, IntDomainVar[] vars, long seed)`
- **type of variable** : integer
- **references** : *n/a*

Example:

10.17 RandomSetVarSelector (Variable selector)

`RandomSetVarSelector`(x) selects a set variable randomly (**dynamic**). The random `seed` can be fixed.

- **Constructor** :
  - `RandomSetVarSelector(Solver s)`
  - `RandomSetVarSelector(Solver s, long seed)`
  - `RandomSetVarSelector(Solver s, SetVar[] vars, long seed)`
- **type of variable** : set
- **references** : *n/a*

Example:

10.18 StaticSetVarOrder (Variable selector)

`StaticSetVarOrder`(x) selects the set variables in the order they appear in array x (**static**).

$$x_1$$

- **Constructor** :
  - `StaticSetVarOrder(Solver s)`
  - `StaticSetVarOrder(Solver s, SetVar[] vars)`
- **type of variable** : set
- **references** : *n/a*

Example:

10.19 StaticVarOrder (Variable selector)

`StaticVarOrder`(x) selects the integer variables in the order they appear in array x (**static**).

$$x_1$$

- **Constructor** :
  - `StaticVarOrder(Solver s)`
  - `StaticVarOrder(Solver s, IntDomainVar[] vars)`
- **type of variable** : integer
- **references** : *n/a*

Example:

Chapter 11

Value iterators (Solver)

This section lists and details the value iterators currently available in Choco.

11.1 DecreasingDomain (Value iterator)

`DecreasingDomain` selects the integer variable largest value:

$$\max(D(x))$$

- **Constructor** : `DecreasingDomain()`
- **type of variable** : integer

11.2 IncreasingDomain (Value iterator)

`IncreasingDomain` selects the integer variable smallest value:

$$\min(D(x))$$

- **Constructor** : `IncreasingDomain()`
- **type of variable** : integer

11.3 RealIncreasingDomain (Value iterator)

`RealIncreasingDomain` selects the real variable smallest value:

$$\min(D(x))$$

- **Constructor** : `RealIncreasingDomain()`

- **type of variable** : real

Chapter 12

Value selector (Solver)

This section lists and details the value selectors currently available in Choco.

12.1 BestFit (Value selector)

`BestFit`, associated with a pack(items, load, bin, size) constraint, selects the bin v with the minimum residual space $rcap_v$ for the bin assignment integer variable $\texttt{bin}[i]$ of item i:

$$v \in D(\texttt{bin}[i]) : \quad \min\{rcap_v\} \qquad \text{with } rcap_v = \overline{\texttt{load}[v]} - \sum_{j \in Ker(\texttt{items}[v])} \texttt{size}[j]$$

- **Constructor** : `BestFit(PackSConstraint cstr)`
- **type of variable** : integer (*bin-to-item assignment variable* $\texttt{bin}[i]$)

12.2 CostRegularValSelector (Value selector)

`CostRegularValSelector`, associated with a `CostRegular`$(z, \langle x_1, .., x_n\rangle, \mathcal{L}(\Pi), \langle c_{i,j}\rangle)$ constraint, selects a value v_i in the domain of the sequence integer variable x_i that can be extended to a solution for the constraint $\langle v_1, .., v_n\rangle \in \mathcal{L}(\Pi)$ of maximum cost (if `max`, resp. minimum cost if $\neg$`max`):

$$v_i \in D(x_i) : \quad \exists v_j \in D(x_j)\ \forall j \neq i, \text{ s.t. } \sum_{j=1}^{n} c_{j,v_j} = \bar{z} \text{ (resp. } = \underline{z})$$

Remember that the underlying data structure of a `CostRegular` constraint is a layered acyclic digraph, where each arc in the i-th layer corresponds to a possible assignment of variable x_i to a value $v \in D(x_i)$ and has length c_{iv}, and where each path corresponds to a solution of the constraint, i.e. a word $\langle v_1, .., v_n\rangle$ of language $\mathcal{L}(\Pi)$, whose cost z is equal to the path length $\sum_{j=1}^{n} c_{j,v_j}$.

This heuristic works with the `ConstraintType.COSTREGULAR` implementation of the constraint: see FCostRegularValSelector for use with the `ConstraintType.FASTCOSTREGULAR` implementation (preferred).

- **Constructor** : `CostRegularValSelector(CostRegular cr, boolean max)`
- **type of variable** : integer

12.3 FCostRegularValSelector (Value selector)

`FCostRegularValSelector`, associated with a CostRegular$(z, \langle x_1, .., x_n \rangle, \mathcal{L}(\Pi), \langle c_{i,j} \rangle)$ constraint, selects a value v_i in the domain of the sequence integer variable x_i that can be extended to a solution for the constraint $\langle v_1, .., v_n \rangle \in \mathcal{L}(\Pi)$ of maximum cost (if `max`, resp. minimum cost if ¬`max`):

$$v_i \in D(x_i) : \quad \exists v_j \in D(x_j)\ \forall j \neq i,\ \text{s.t.} \sum_{j=1}^{n} c_{j,v_j} = \overline{z}\ (\text{resp.}\ = \underline{z})$$

Remember that the underlying data structure of a `CostRegular` constraint is a layered acyclic digraph, where each arc in the i-th layer corresponds to a possible assignment of variable x_i to a value $v \in D(x_i)$ and has length c_{iv}, and where each path corresponds to a solution of the constraint, i.e. a word $\langle v_1, .., v_n \rangle$ of language $\mathcal{L}(\Pi)$, whose cost z is equal to the path length $\sum_{j=1}^{n} c_{j,v_j}$.

This heuristic works with the `ConstraintType.FASTCOSTREGULAR` implementation of the constraint (preferred): see CostRegularValSelector for use with the `ConstraintType.COSTREGULAR` implementation.

- **Constructor** : `FCostRegularValSelector(FastCostRegular cr, boolean max)`
- **type of variable** : integer

12.4 MaxVal (Value selector)

`MaxVal` selects the largest value in the domain of the integer variable:

$$\max(D(x))$$

- **Constructor** : `MaxVal()`
- **type of variable** : integer

12.5 MidVal (Value selector)

`MidVal` selects the closest value (equal or greater) to the integer variable domain midpoint:

$$\min(v \mid v \geq \frac{\underline{x} + \overline{x}}{2})$$

- **Constructor** : `MidVal()`
- **type of variable** : integer

12.6 MinEnv (Value selector)

`MinEnv` selects the smallest value in the open domain of the set variable:

$$\min(Env(x) \setminus Ker(x))$$

- **Constructor** : `MinEnv()`
- **type of variable** : set

12.7 MinVal (Value selector)

`MinVal` selects the smallest value in the domain of the integer variable:

$$\min(D(x))$$

- **Constructor** : `MinVal()`
- **type of variable** : integer

12.8 RandomSetValSelector (Value selector)

`RandomSetValSelector` selects a random value in the open domain of the set variable. The random `seed` can be specified:

$$rand(Env(x) \setminus Ker(x))$$

- **Constructor** :
  - `RandomSetValSelector()`
  - `RandomSetValSelector(long seed)`
- **type of variable** : set

Part III

Extras

Chapter 13

Choco and Visu

13.1 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*.

13.2 The visu package

The *visu* package contains objects to define a visualization of the resolution, domain reduction, constraints propagation, etc.

Figures 13.1 depicts the class diagram of the visu package (*powered by BOUML*):

13.3 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.

13.3.1 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:

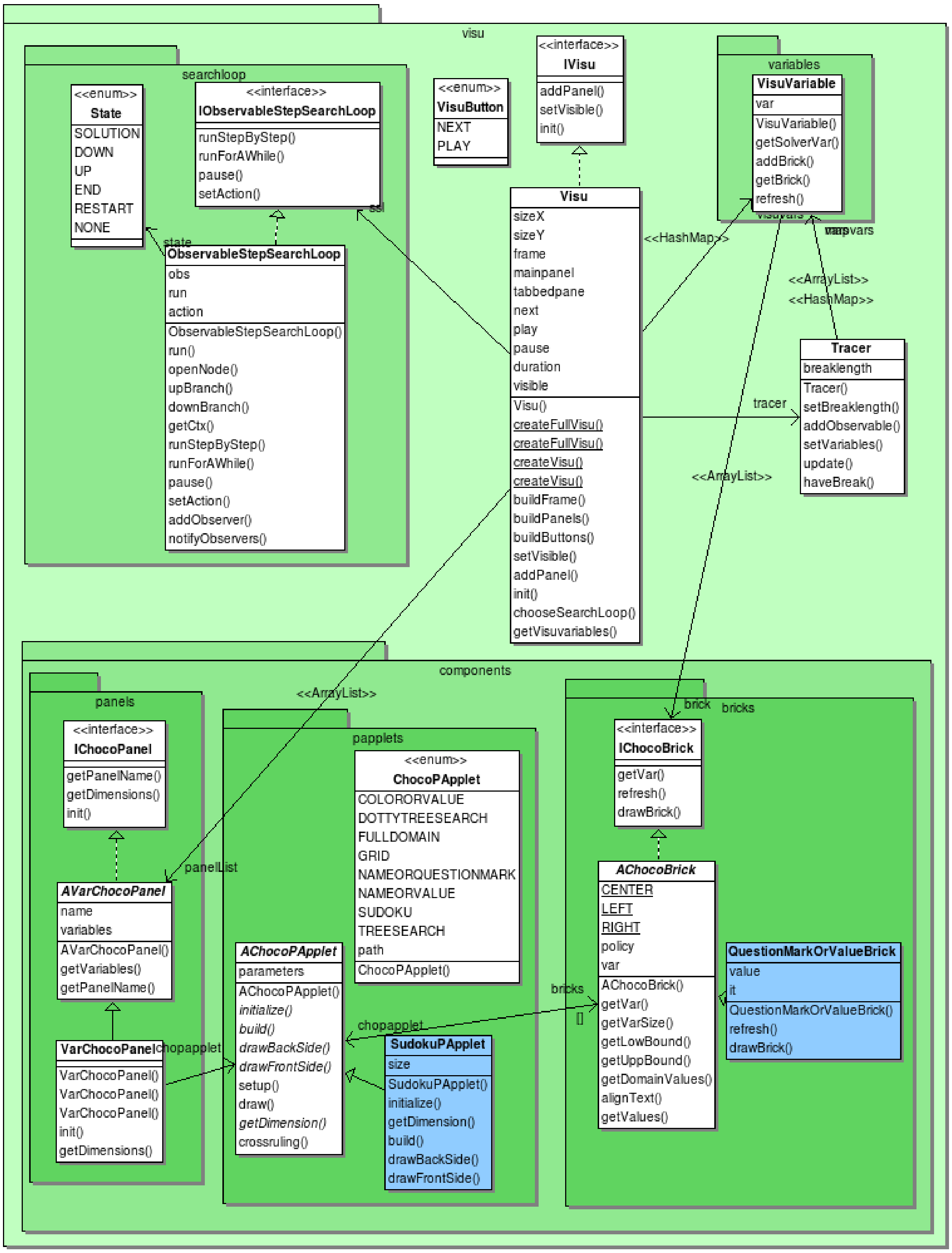


Figure 13.1: Visu classes diagram. The blue classes are examples of implementation and inheritence.

```
Visu v = Visu.createVisu();
```

13.3.2 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

13.4 Examples

UNDER DEVELOPMENT

Chapter 14

Sudoku and Constraint Programming

14.1 Sudoku ?!?

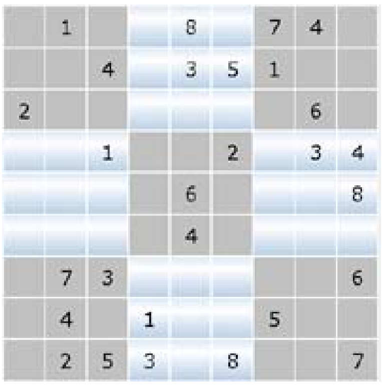


Figure 14.1: A sudoku grid

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.

14.1.1 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.

14.2 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.

14.2.1 Simple rules: single candidate and single position

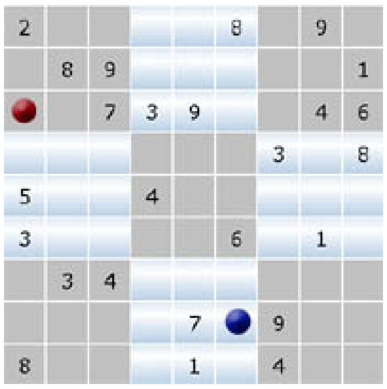


Figure 14.2: Simple rules: single candidates and single position

Let consider the grid on Figure 14.2 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 !

14.2.2 Human reasoning principles

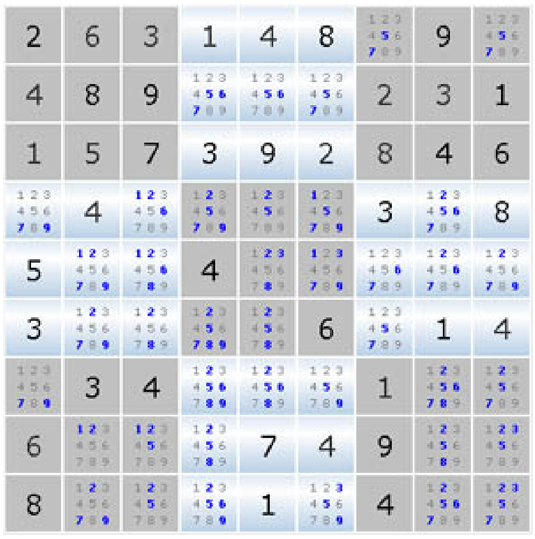
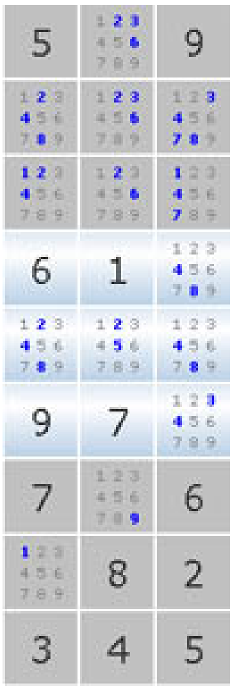


Figure 14.3: Introducing marks

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

14.2.3 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.

14.3 See also

- SudokuHelper a sudoku solver and helper applet developed with *Choco*.
- PalmSudoku a rather complete list of rules and tips for solving sudokus

Glossary

branching strategy heuristic controlling the execution of a search loop at a point where the control flow may be split between different branches. 29

search strategy composition of branching strategies. 29

Solver solver description. 27

value iterator heuristic specifying how to choose a value from a chosen variable, through an iterator, at a fix point. 29

value selector heuristic specifying how to choose dynamically a value from a chosen variable at a fix point. 29

variable selector heuristic specifying how to choose a variable at a fix point. 29

Bibliography

[Beldiceanu and Carlsson, 2002] Beldiceanu, N. and Carlsson, M. (2002). A new multi-resource `cumulatives` constraint with negative heights. In *International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*, pages 63–79. Springer-Verlag.

[Beldiceanu et al., 2005] Beldiceanu, N., Carlsson, M., Debruyne, R., and Petit, T. (2005). Reformulation of global constraints based on constraint checkers. *Constraints*, 10(4):339–362.

[Beldiceanu et al., 2008] Beldiceanu, N., Flener, P., and Lorca, X. (2008). Combining tree partitioning, precedence, and incomparability constraints. *Constraints*, 13(4):459–489.

[Bessière et al., 2005a] Bessière, C., Hebrard, E., Hnich, B., Kızıltan, Z., and Walsh, T. (2005a). *Among*, *common* and *disjoint* Constraints. In Carlsson, M., Fages, F., Hnich, B., and Rossi, F., editors, *Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2005)*, pages 223–235.

[Bessière et al., 2005b] Bessière, C., Hebrard, E., Hnich, B., Kızıltan, Z., and Walsh, T. (2005b). Filtering algorithms for the `nvalue` constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'05)*, volume 3524 of *LNCS*, pages 79–93. Springer-Verlag.

[Bessière et al., 2006] Bessière, C., Hebrard, E., Hnich, B., Kızıltan, Z., and Walsh, T. (2006). *Among*, *common* and *disjoint* Constraints. In Hnich, B., Carlsson, M., Fages, F., and Rossi, F., editors, *Recent Advances in Constraints, Joint ERCIM/Colognet International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP05; Uppsala, Sweden, June 2005; Revised Selected and Invited Papers*, volume 3978 of *LNAI*, pages 28–43. Springer-Verlag.

[Bleuzen-Guernalec and Colmerauer, 1997] Bleuzen-Guernalec, N. and Colmerauer, A. (1997). Narrowing a block of sortings in quadratic time. In *International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume ? of *LNCS*, pages ?–? Springer-Verlag.

[Boussemart et al., 2004] Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004). Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 146–150. IOS Press 2004.

[Carlsson and Beldiceanu, 2002] Carlsson, M. and Beldiceanu, N. (2002). Arc-consistency for a chain of lexicographic ordering constraints. Technical Report T2002-18, Swedish Institute of Computer Science.

[Demassey et al., 2006] Demassey, S., Pesant, G., and Rousseau, L.-M. (2006). A `Cost-Regular` based hybrid column generation approach. *Constraints*, 11(4):315–333. Special issue following CPAIOR'05.

[Frisch et al., 2002] Frisch, A. M., Hnich, B., Kızıltan, Z., Miguel, I., and Walsh, T. (2002). Global constraints for lexicographic orderings. In *International Conference on Principles and Practice of Constraint Programming (CP'2002)*, volume 2470 of *LNCS*, pages 93–108. Springer-Verlag.

[Frisch et al., 2003] Frisch, A. M., Hnich, B., Kızıltan, Z., Miguel, I., and Walsh, T. (2003). Multiset ordering constraints. In *18th International Joint Conference on Artificial Intelligence (IJCAI'03)*.

[López-Ortiz et al., 2003] López-Ortiz, A., Quimper, C.-G., Tromp, J., and van Beek, P. (2003). A fast and simple algorithm for bounds consistency of the `alldifferent` constraint. In *18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 245–250.

[Mehlhorn and Thiel, 2000] Mehlhorn, K. and Thiel, S. (2000). Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *International Conference on Principles and Practice of Constraint Programming (CP'00)*, volume ? of *LNCS*, pages ?–? Springer-Verlag.

[Menana and Demassey, 2009] Menana, J. and Demassey, S. (2009). Sequencing and counting with the `multicost-regular` constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 of *LNCS*, pages 178–192. Springer-Verlag.

[Pesant, 2004] Pesant, G. (2004). A regular language membership constraint for finite sequences of variables. In *International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *LNCS*, pages 482–495. Springer-Verlag.

[Quimper et al., 2003] Quimper, C.-G., van Beek, P., López-Ortiz, A., Golynski, A., and Sadjad, S. B. (2003). An efficient bounds consistency algorithm for the *global-cardinality* constraint. In *International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 of *LNCS*, pages 600–614. Springer-Verlag.

[Refalo, 2004] Refalo, P. (2004). Impact-based search strategies for constraint programming. In *International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *LNCS*, pages 557–570. Springer-Verlag.

[Régin, 1994] Régin, J.-C. (1994). A filtering algorithm for constraints of difference in CSP. In *12th National Conference on Artificial Intelligence (AAAI'94)*, pages 362–367.

[Régin, 1996] Régin, J.-C. (1996). Generalized arc consistency for `global-cardinality` constraint. In *14th National Conference on Artificial Intelligence (AAAI'96)*, pages 209–215.

[Shaw, 2004] Shaw, P. (2004). A constraint for bin packing. In *International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *LNCS*, pages 648–662. Springer-Verlag.

[Trick, 2003] Trick, M. A. (2003). A dynamic programming approach for consistency and propagation for knapsack constraints. In Netherlands, S., editor, *Annals of Operations Research*, volume Volume 118, Numbers 1-4 / feb. 2003, pages 73–84. Springer Netherlands.

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

2. VERBATIM COPYING

3. COPYING IN QUANTITY

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS