choco-tuto Documentation

Release 1.0.0

Charles Prud'homme

Sep 14, 2017

Contents

CHAPTER 1

Introduction

Introduction to choco-tuto

author Charles Prud'homme

Presentation

I'm the project leader of choco-solver since 2008.

It has been a while since I first though of writing a tutorial. Now that a new version of Choco, namely 4.0.0, is being released, the time has come to jump in.

This tutorial is written using RST like a private lesson.

If needed, you will find a brief introduction to Constraint programming.

Let's go for a ride now.

About Constraint Programming

Caution: In this documentation, it is assumed that the reader is already familiar with the theoretical concepts of constraint-programming. The current material is meant to help the reader to learn the basics of Choco 4.0.0, not constraint-programming itself.

General documentation on constraint-programming may be found at:

- Wikipedia
- Handbook of Constraint Programming
- Hakan's advices
- Bartak's courses

- MOOC

In case the reader is completely new to the domain of constraint-programming, it is highly recommended to attend training courses.

Choco, a constraint programming java library

What is Choco ?

Choco is a Free and Open-Source Java library dedicated to Constraint Programming. It aims at describing hard combinatorial problems in the form of Constraint Satisfaction Problems and solving them with Constraint Programming techniques.

A little bit of history

The first version of Choco was written in 1999 by François Laburthe and Narendra Jussien in Claire. The main objective was to provide an open source library for constraint programming dedicated to teaching and research.

In 2003, the code was then converted to Java by Hadrien Cambazard, Guillaume Rochart. Funded by EMN, Bouygues and Amadeus, I was recruited as project leader in 2008. Together with Hadrien Cambazard and Arnaud Malapert, we developed and maintained Choco-v2.

It was then time for a deep refactoring, with main objective to clean up the code and focus on improving resolution performances. A joint work of Jean-Guillaume Fages and I, with the system expertise of Narendra Jussien and Xavier Lorca, led to Choco-v3, first released in 2011. This version won seven medals in four entries in the MiniZinc Challenge.

More recently, Jean-Guillaume and I wanted to review the API and the last version was born.

Why “Choco” ?

Choco is the acronym of the french sentence: “Chouette, une boite à Outils de Contraintes orientée Object” which can be translated into: “Great, an object-oriented constraint toolbox”.

Why using Choco ?

Choco is one of the more mature free open source Java library for constraint programming. In a nutshell:

- three types of variables: integer, boolean and set,
- more than 70 constraints : the most useful and state-of-the-art implementations,
- PLM framework allows configurable searches and most wide-spread search strategies,
- deal with satisfaction and optimization (mono and multi) problems, multi-thread resolution,

But also:

- explanations supported,
- a MiniZinc and XCSP3 instance parser,
- graph variables/constraints available,
- and gentle binding to Ibex: real variables and constraints supported

If that doesn't sound like something familiar to you, let's give a try !

Installation

Requirements

You need to have JDK 8 installed on your working environment, and except if you are an emacs master, an Integrated Development Environment would be needed.

If you just want to try Choco wihtout installing anything, you can have a look at the Choco online IDE.

Download

The JAR anyone can start with is named *choco-solver-X.y.z-with-dependencies.jar* where *X.y.z* denotes the version you want to use, here, 4.0.0. It contains the constraint programming API and any needed dependencies.

Choco is available on the official website and on Maven Central Repository.

Installation

Since Choco is a Java library, it does not need to installed strictly speaking. You only have to add it to the classpath of your project.

See instructions for IntelliJ IDEA or Eclipse.

As a Maven dependency

Choco is available on the official website and, thus, you can edit your *pom.xml* with :

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

where *X.y.z* denotes the version you want to use, here, 4.0.0.

Building from sources

The source of the released versions are directly available on GitHub. You will then need Git and Maven 3+ on your working environment.

Check everything is alright

To make sure you correctly configured your project (with Java 8 and Choco), create a new class and copy/paste the following code:

```
public static void main(String[] args) {
    Model model = new Model("A first model");
    System.out.println(model.getName());
}
```

and execute it. The console should output :

```
A first model
```

CHAPTER 2

Through a first example

A first example

Modelling the eight queens puzzle

First of all, let's consider the eight queen puzzle, frequently used to introduce constraint programming.

Wikipedia told us that:

> The eight queens puzzle is the problem of placing eight chess queens on an 8x8 chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

The problem can be generalized to the *n*-queens problem (placing *n* queens on a nxn chessboard).

There are many ways to model this problem with Choco, we will start with a basic one:

```
int n = 8;
Model model = new Model(n + "-queens problem");
IntVar[] vars = new IntVar[n];
for(int q = 0; q < n; q++){
    vars[q] = model.intVar("Q_"+q, 1, n);
}
for(int i  = 0; i < n-1; i++){
    for(int j = i + 1; j < n; j++){
        model.arithm(vars[i], "!=",vars[j]).post();
        model.arithm(vars[i], "!=", vars[j], "-", j - i).post();
        model.arithm(vars[i], "!=", vars[j], "+", j - i).post();
    }
}
Solution solution = model.getSolver().findSolution();
if(solution != null){
    System.out.println(solution.toString());
}
```

If you copy/paste the this code and execute it, it prints the value that each variable takes in the solution on the console

```
Solution: Q_0=7, Q_1=4, Q_2=2, Q_3=5, Q_4=8, Q_5=1, Q_6=3, Q_7=6,
```

Now, let's discuss the code.

The model

On line 2, a *model* is declared. It is the key component of the library and needed to describe any problem.

The variables

A queen position is defined by its coordinates on the chessboard. Naturally, we don't know yet where to put queens on the chessboard, but we can give indications. To do so, we need to declare *variables*.

A variable is an *unknown* which has to be assigned to value in a solution. The values a variable can take is defined by its domain.

Here, in a solution, there will be exactly one queen per row (and per column). So, a modelling trick is to fix the row a queen can go to and only question on their column. Thus, there will be *n* queens (one per row), each of them to be assigned to one column, among $[1, n]$.

Lines 3 and 5 managed to create variables and their domain.

The constraints

The queens' position must follow some rules. We already encoded that there can only be one queen per row. Now, we have to ensure that, on any solution, no two queens share the same column and diagonal.

First, the columns conditions: if the queen *i* is on column *k*, then any other queens cannot take the value *k*. So, for each pair of queens, the two related variables cannot be assigned to the same value. This is expressed by the *constraint* on line 9. To activate the constraint, it has to be *posted*.

Second, the diagonals: we have to consider the two orthogonal diagonals. If the queen *i* is on column *k*, then, the queen *i+1* cannot be assigned to *k+1*. More generally, the queen *i+m* cannot be assigned to *k+m*. The same goes with the other diagonal. This is declared on line 10 and 11.

Solving the problem

Once the problem has been described into a model using variables and constraints, its satisfaction can be evaluated, by trying to *solve* it.

This is achieved on line 14 by calling the *getSolver().findSolution()* method from the model. If a solution exists, it is printed on the console

What to do next ?

We are going to use and extend this small problem in the future. But before, we will have a look at the different objects we can manipulate.

Objects in Choco

The content of this section is extracted from the Javadoc and the Choco's User Guide. Here, we **briefly** described the main aspects of the most commonly used objects. This does not aim at being complete: it covers the basic information.

The *Model*

```
Model model = new Model("My model");
```

As said before, the *Model* is a key component of the library. It has to be the first instruction declared, since it provides entry point methods that help modelling a problem.

A good habit is to declare a model with a name, otherwise a random one will be assigned by default.

We designed the model in such a way that you can reach almost everything needed to describe a problem from it.

For example, it stores its variables and constraints. Variables and constraints of a model can be retrieved thanks to API :

```
model.retrieveIntVars(true); // extract IntVars, including BoolVars
model.getCstrs(); // extract posted constraints
```

Note: We strongly encourage you to attach the Javadoc (provides either on the website or on Maven Central Repository) to the library in your IDE.

The *Variables*

A variable is an *unknown*, mathematically speaking. In a solution of a given problem (considering that at least one exists), each variable is assigned to a *value* selected within its domain. The notion of *value* differs from one type of variable to the other.

Note: A variable can be declared in only one model at a time. Indeed, a reference to the declaring model is maintained in it.

Integer variable

An integer variable, *IntVar*, should be assigned to an integer. There are many ways to declare an *IntVar*

```
// A variable with a unique value in its domain, in other words, a constant
IntVar two = model.intVar("TWO", 2);
// Any value in [1..4] can be assigned to this variable
IntVar x = model.intVar("X", 1, 4);
// Only the values 1, 3 and 4 can be assigned to this variable
IntVar y = model.intVar("X", new int[]{1, 3, 4});
```

Caution: Declaring a variable with an *infinite* domain, like :

```
model.intVar("X", Integer.MIN_VALUE, Integer.MAX_VALUE)
```

is clearly a bad idea.

Too large domains may lead to underflow or overflow issues and most of the time, even if Choco will finally compute the right bounds by itself, you certainly want to save space and time by directly declaring relevant bounds.

The domain of an integer variable in Choco can either be *bounded* or *enumerated*. In a bounded domain, only current bounds are stored in memory. This saves memory (only two integers are needed) but it restricts its usage: there is no possibility to make holes in it.

On the contrary, with an enumerated domain, all possible values are explicitly stored in memory. This consumes more memory (one integer and a bitset – many longs – are needed) but it allows making holes in it.

Modelling: Bounded or Enumerated?

The memory consumption should not be the only criterion to consider when one needs to choose between one representation and the other. Indeed, the *filtering* strength of the model, through constraints, has to be considered too. For instance, some constraints can only deduce bound updates, in that case bounded domains fit the need. Other constraints can make holes in variables' domain, in that case enumerated domains are relevant.

If you don't know what to do, the following scenario can be applied:

- domain's cardinality greater than 262144 should be bounded
- domain's cardinality smaller than 32768 can be enumerated without loss of efficiency
- in any case, empirical evaluation is a good habit.

Boolean variable

An boolean variable, *BoolVar*, should be assigned to a boolean. A *BoolVar* is a specific *IntVar* with a domain restricted to $[0, 1]$, 0 stands for *false*, 1 for *true*. Thus a *BoolVar* can be declared in any integer constraint (*e.g.*, a sum) and boolean constraints (*e.g.*, in clauses store).

Here is the common way to declare a *BoolVar*

```
// A [0,1]-variable
BoolVar b = model.boolVar("b");
```

Set variable

A set variable, *SetVar*, should be assigned to a set of integers (possibly empty or singleton). Its domain is defined by a set of intervals *[LB,UB]* where *LB* denotes the integers that figure in all solutions and *UB* the integers that potentially figure in a solution.

```
// SetVar representing a subset of {1,2,3,5,12}
SetVar y = model.setVar("y", new int[]{}, new int[]{1,2,3,5,12});
// possible values: {}, {2}, {1,3,5} ...
```

Real variable

A real variable, *RealVar*, should be assigned an interval of doubles. Its domain is defined by its bounds and a *precision*. The precision parameter helps considering a real variable as instantiated: when the distance between the two bounds is less than or equal to the precision.

```
// A [0.2d, 3.4d]-variable, with a precision of 0.001d
RealVar x = model.realVar("x", 0.2d, 3.4d, 0.001d);
```

Note: Using *RealVar* requires to install Ibex before. Indeed, Choco relies on Ibex to deal with continuous constraints.

The *Constraints*

A constraint is a relation between one or more variables of a model. It defines conditions over these variables that must be respected in a solution. A constraint has a semantic (*e.g.*, "greater than" or "all different") and is equipped with *filtering algorithms* that ensure conditions induced by the semantic hold.

A filtering algorithm, or *propagator*, removes from variables' domain values that cannot appear in any solution. A propagator has a *filtering strength* and a time complexity to achieve it. The filtering strength, or *level of consistency*, determines how accurate a propagator is when values to be removed are detected.

Posting a constraint

For a constraint to be integrated in a model, a call to *post()* is required :

```
// x and y must be different in any solution
model.arithm(x, "!=", y).post();
// or, in a more verbose way
model.post(model.arithm(x, "<", z));
```

Note: A constraint can be posted in only one model at a time. Indeed, a reference to the declaring model is maintained in it.

Once posted, a constraint is known from a model and will be integrated in the filtering loop.

Note: Posting a constraint does not remove any value from its variables' domain. Indeed, Choco runs the *initial propagation* only when a resolution is called.

The only reason why a constraint is not posted a model is to *reify* it.

Reifying a constraint

Alternatively, a constraint can be reified with a *BoolVar* :

```
// the constraint is reified with `b`
BoolVar r1 = model.arithm(x, "!=", y).reify();
// equivalent to:
BoolVar r2 = model.boolVar("r2");
model.arithm(x, "<", z).reifyWith(r2);
```

The *BoolVar* that reifies a constraint represents whether or not a constraint is satisfied. If the constraint is satisfied, the boolean variable is set to *true*, *false* otherwise. If the boolean variable is set to *true* the constraint should be satisfied, unsatisfied otherwise.

Reifying constraints is helpful to express conditions like: *(x = y) xor (x > 15)* :

```
BoolVar c1 = model.arithm(x, "=", y).reify();
BoolVar c2 = model.arithm(x, ">", 15).reify();
model.arithm(c1, "+", c2, "=", 1).post();
```

Warning: A reified constraint **should not** be posted. Indeed, posting it will declare it as a *hard* constraint, to be satisfied, reifying it will declare it as a *soft* constraint, that can be unsatisfied. Both state cannot co-exist simultaneously: hard state dominates soft one.

Caution: A constraint that is neither posted or reified **is not considered at all** in the resolution. Make sure all constraints are either posted or reified.

There are more than 80 constraints in Choco, and anyone can create its own constraint easily. Native constraints are provided by the model, as seen before. A look at the Javadoc gives a big picture of the ones available. In this tutorial, we will have a look at the most commonly used ones.

The *Solver*

The *Model* serves at describing the problem with variables and constraints. The resolution is managed by the *Solver*.

```
Model model = new Model("My problem");
// variables declaration
// constraints declaration
Solver solver = model.getSolver();
Solution solution = solver.findSolution();
```

Having access to the *Solver* is needed to tune the resolution and launch it. It provides methods to configure *search strategies*, to define resolution goals (*i.e.*, finding one solution, all solutions or optimal solutions) and getting resolution statistics.

Instead of listing all resolution features, we will see some of them in the following.

Modelling and Solving

Carefully selecting variables and constraints to describe a problem in a model is a tough task to do. Indeed, some knowledge of the available constraints (or their reformulations), their filtering strength and complexity, is needed to take advantage of Constraint Programming. This has to be both taught and experimented. Same goes with the resolution tuning. Using Choco has a black-box solver results in good performance on average. But, injecting problem expertise in the search process is a key component of success. Choco offers a large range of features to let you good chances to master your problem.

Playing around with 8-queens puzzle

We will now see and comment some modifications of the code presented previously :

```
1 int n = 8;
2 Model model = new Model(n + "-queens problem");
3 IntVar[] vars = new IntVar[n];
4 for(int q = 0; q < n; q++){
```

```
5       vars[q] = model.intVar("Q_"+q, 1, n);
6   }
7   for(int i  = 0; i < n-1; i++){
8       for(int j = i + 1; j < n; j++){
9           model.arithm(vars[i], "!=",vars[j]).post();
10          model.arithm(vars[i], "!=", vars[j], "-", j - i).post();
11          model.arithm(vars[i], "!=", vars[j], "+", j - i).post();
12      }
13  }
14  Solution solution = model.getSolver().findSolution();
15  if(solution != null){
16      System.out.println(solution.toString());
17  }
```

Variables

First, lines 3-6 can be compacted into:

```
IntVar[] vars = model.intVarArray("Q", n, 1, n, false);
```

Doing so, an n-array of variables with [1,n]-domain is created. Each variable name is "Q[i]" where *i* is its position in the array, starting from 0. The last parameter, set to false, indicates that the domains must be enumerated (not bounded).

Constraints

Second, lines 9 to 11 can be replaced by:

```
vars[i].ne(vars[j]).post();
vars[i].ne(vars[j].sub(j - i)).post();
vars[i].ne(vars[j].add(j - i)).post();
```

where *ne* stands for *not equal*. Theses instructions express the same constraints, or more complex expressions, in a convenient way. Here the expression is posted as a decomposition: the AST is analyzed and additional variables and constraints are added on the fly.

Note: Calling `e.post()` on an expression *e* is a syntactic sugar for `e.decompose().post()`.

Alternatively, one can decide to generate the possible combinations from the expression and post *table* constraints[1]. To do so, the expression should be first turned into extension constraint then be posted

```
vars[i].ne(vars[j]).extension().post();
vars[i].ne(vars[j].sub(j - i)).extension().post();
vars[i].ne(vars[j].add(j - i)).extension().post();
```

Global constraints

Here we posted three groups of 28 constraints. The first group expresses that two queens cannot be on the same column by posting a *clique* of inequality constraints. The second and third groups express the same conditions for each diagonal.

[1] such constraint are defined by a set of allowed/forbidden tuples.

In other words, the variables of each groups must be *all different*. Luckily, there exists a *global constraint* that captures that conditions:

> *Global constraints specify patterns that occur in many problems and exploit efficient and effective constraint propagation algorithms for pruning the search space.*
>
> ("*The Complexity of Global Constraints*", C.Bessière, E.Hebrard, B.Hnich and T.Walsh, AAAI 2004.)

We can reformulate the set of constraints to:

```
IntVar[] diag1 = new IntVar[n];
IntVar[] diag2 = new IntVar[n];
for(int i = 0 ; i < n; i++){
    diag1[i] = vars[i].sub(i).intVar();
    diag2[i] = vars[i].add(i).intVar();
}
model.post(
    model.allDifferent(vars),
    model.allDifferent(diag1),
    model.allDifferent(diag2)
);
```

Or with Java8 lambdas:

```
IntVar[] diag1 = IntStream.range(0, n)
                          .mapToObj(i -> vars[i].sub(i).intVar())
                          .toArray(IntVar[]::new);
IntVar[] diag2 = IntStream.range(0, n)
                           .mapToObj(i -> vars[i].add(i).intVar())
                           .toArray(IntVar[]::new);
model.post(
    model.allDifferent(vars),
    model.allDifferent(diag1),
    model.allDifferent(diag2)
);
```

The constraint on line 8 simply states that all variables from *vars* must be different. The constraint on line 9 (or 10) states that all variables from a diagonal must be different. The variables of a diagonal are given by `IntStream.range(0, n).mapToObj(i -> vars[i].add(i).intVar()).toArray(IntVar[]::new)` (lines 4-6) which construct an array of *IntVar* from the mapping function `-> vars[i].add(i).intVar()`. This function maps each index *i* in the [0,n] range to an integer variable equals to `vars[i].add(i)`. The call to the *intVar()* method effectively turns the arithmetic expression into an integer variable. This extraction may introduce additional variables and constraints automatically.

Solver

To compare the first model and the modified one, we need to get features and measures. A call to *solver.showStatistics();* will output commonly used indicators to the console, such as the number of variables, constraints, solutions found, open nodes, etc.

We can either let the solver explore the search space by itself or define a search strategy, like:

```
solver.setSearch(Search.domOverWDegSearch(vars));
```

Updated code

```
int n = 8;
Model model = new Model(n + "-queens problem");
IntVar[] vars = model.intVarArray("Q", n, 1, n, false);
IntVar[] diag1 = IntStream.range(0, n).mapToObj(i -> vars[i].sub(i).intVar()).
↪toArray(IntVar[]::new);
IntVar[] diag2 = IntStream.range(0, n).mapToObj(i -> vars[i].add(i).intVar()).
↪toArray(IntVar[]::new);
model.post(
    model.allDifferent(vars),
    model.allDifferent(diag1),
    model.allDifferent(diag2)
);
Solver solver = model.getSolver();
solver.showStatistics();
solver.setSearch(Search.domOverWDegSearch(vars));
Solution solution = solver.findSolution();
if (solution != null) {
    System.out.println(solution.toString());
}
```

Running the following code outputs something like:

```
** Choco 4.0.0 (2016-05) : Constraint Programming Solver, Copyleft (c) 2010-2016
- Model[8-queens problem] features:
    Variables : 32
    Constraints : 19
    Default search strategy : no
    Completed search strategy : no
1 solution found.
    Model[8-queens problem]
    Solutions: 1
    Building time : 0,000s
    Resolution time : 0,012s
    Nodes: 6 (491,9 n/s)
    Backtracks: 0
    Fails: 0
    Restarts: 0
    Variables: 32
    Constraints: 19
Solution: Q[0]=7, Q[1]=4, Q[2]=2, Q[3]=8, Q[4]=6, Q[5]=1, Q[6]=3, Q[7]=5,
```

Basically, the trace informs that:

- there are 32 variables: the eight queens, and the additional ones induced by expressions extraction,
- there are 19 constraints: three *allDifferent* constraints, and the additional ones induced by expressions extraction,
- one solution has been found,
- it took 11 ms to find it,
- in the meantime, 6 decisions were made and none of them were wrong.

CHAPTER 3

Practicals

Verbal arithmetic

A **verbal arithmetic** is a mathematical equation among unknown numbers, whose digits are represented by letters (see wikipedia for more details).

Input data

The equation we consider here is:

```
    S E N D
+   M O R E
-----------
= M O N E Y
```

A value has to be assigned to each letter in such a way that the equation is satisfied. Note that:

- each letter corresponds to a digit,
- a word cannot start with à 0,
- no two letters are assigned to the same digit.

mathematical model>>

Warehouse Location Problem

In the Warehouse Location problem (WLP), a company considers opening warehouses at some candidate locations in order to supply its existing stores.

Each possible warehouse has the same maintenance cost, and a capacity designating the maximum number of stores that it can supply.

Each store must be supplied by exactly one open warehouse. The supply cost to a store depends on the warehouse.

The objective is to determine which warehouses to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is minimized.

See this page for more details.

Input data

We consider here the following input (in java):

```
// number of warehouses
int W = 5;
// number of stores
int S = 10;
// maintenance cost
int C = 30;
// capacity of each warehouse
int[] K = new int[]{1, 4, 2, 1, 3};
// matrix of supply costs, store x warehouse
int[][] P = new int[][]{
    {20, 24, 11, 25, 30},
    {28, 27, 82, 83, 74},
    {74, 97, 71, 96, 70},
    {2, 55, 73, 69, 61},
    {46, 96, 59, 83, 4},
    {42, 22, 29, 67, 59},
    {1, 5, 73, 59, 56},
    {10, 73, 13, 43, 96},
    {93, 35, 63, 85, 46},
    {47, 65, 55, 71, 95}};
```

mathematical model>>

Aircraft Landing Problem

Given a set of planes and runways, the objective is to minimize the total (weighted) deviation from the target landing time for each plane.

There are costs associated with landing either earlier or later than a target landing time for each plane.

Each plane has to land on one of the runways within its predetermined time windows such that separation criteria between all pairs of planes are satisfied.

This type of problem is a large-scale optimization problem, which occurs at busy airports where making optimal use of the bottleneck resource (the runways) is crucial to keep the airport operating smoothly.

See this page for more details.

Input data

We consider here the following input (in java):

```
// number of planes
int N = 10;
// Times per plane: {earliest landing time, target landing time, latest landing time}
```

```
int[][] LT = {
    {129, 155, 559},
    {195, 258, 744},
    {89, 98, 510},
    {96, 106, 521},
    {110, 123, 555},
    {120, 135, 576},
    {124, 138, 577},
    {126, 140, 573},
    {135, 150, 591},
    {160, 180, 657}};
// penalty cost penalty cost per unit of time per plane: {for landing before target,␣
↪after target}
int[][] PC = {
    {10, 10},
    {10, 10},
    {30, 30},
    {30, 30},
    {30, 30},
    {30, 30},
    {30, 30},
    {30, 30},
    {30, 30},
    {30, 30}};

// Separation time required after i lands before j can land
int[][] ST = {
    {99999, 3, 15, 15, 15, 15, 15, 15, 15, 15},
    {3, 99999, 15, 15, 15, 15, 15, 15, 15, 15},
    {15, 15, 99999, 8, 8, 8, 8, 8, 8, 8},
    {15, 15, 8, 99999, 8, 8, 8, 8, 8, 8},
    {15, 15, 8, 8, 99999, 8, 8, 8, 8, 8},
    {15, 15, 8, 8, 8, 99999, 8, 8, 8, 8},
    {15, 15, 8, 8, 8, 8, 99999, 8, 8, 8},
    {15, 15, 8, 8, 8, 8, 8, 99999, 8, 8},
    {15, 15, 8, 8, 8, 8, 8, 8,  99999, 8},
    {15, 15, 8, 8, 8, 8, 8, 8,  8, 99999}};
```

mathematical model>>

Nonogram

Nonograms are a popular puzzles, which goes by different names in different countries.

Models have to shade in squares in a grid so that blocks of consecutive shaded squares satisfy constraints given for each row and column.

Constraints typically indicate the sequence of shaded blocks (e.g. 3,1,2 means that there is a block of 3, then a gap of unspecified size, a block of length 1, another gap, and then a block of length 2).

See Nonogram for more details.

Input data

We consider here the following input (in java):

```
// sequence of shaded blocks
int[][][] BLOCKS =
        new int[][][]{{
                {2},
                {4, 2},
                {1, 1, 4},
                {1, 1, 1, 1},
                {1, 1, 1, 1},
                {1, 1, 1, 1},
                {1, 1, 1, 1},
                {1, 1, 1, 1},
                {1, 2, 2, 1},
                {1, 3, 1},
                {2, 1},
                {1, 1, 1, 2},
                {2, 1, 1, 1},
                {1, 2},
                {1, 2, 1},
        }, {
                {3},
                {3},
                {10},
                {2},
                {2},
                {8, 2},
                {2},
                {1, 2, 1},
                {2, 1},
                {7},
                {2},
                {2},
                {10},
                {3},
                {2}}};
```

mathematical model>>

Golomb ruler

Wikipedia told us that:

> A **Golomb ruler** is a set of marks at integer positions along an imaginary ruler such that no two pairs of marks are the same distance apart. The number of marks on the ruler is its order, and the largest distance between two of its marks is its length. The objective is to find optimal (minimum length) or near optimal rulers. Translation and reflection of a Golomb ruler are considered trivial, so the smallest mark is customarily put at 0 and the next mark at the smaller of its two possible values.

Input data

Only the order *m* is given as input data.

mathematical model>>

CHAPTER 4

Customize

Creating a constraint

In this part, we are going to see how to create a constraint to be used by Choco. The work will be based on the sum constraint, more specifically: $\sum_{i=1}^{n} x_i \leq b$ where $x_i = [\underline{x_i}, \overline{x_i}]$ are distinct variables and where b is a constant.

Bounds Consistency Techniques for Long Linear Constraint by W.Harvey and J.Schimpf described in details how such a constraint is implemented and will serve as a basis of this tutorials.

Important: The implementation presented here can be improved in many ways but that is not the goal this tutorial to discuss improvements but to show what is important to know when creating a constraint.

The first filtering algorithm they depicted in the article is roughly the following:

- First, compute $F = b - \sum_{i=1}^{n} \underline{x_i}$
- then, update variables domain, $\forall i \in [1, n], x_i \leq F + \underline{x_i}$

Note that if $F < 0$ the constraint is unsatisfiable.

A first implementation

When one needs to declare its own constraint, actually, he needs to create a propagator. Indeed, in choco, a constraint is a container which is composed of propagators, and each propagator has the right to eliminate values from domain variables. So the first step will be to create a java class that extends *Propagator<IntVar>*. The generic parameter *<IntVar>* indicates that the propagator only manages integer variable. Set it to *BoolVar*, *SetVar* or *Variable* are possible alternatives.

Once the class is created, a constructor is needed plus two methods :

- *public void propagate(int evtmask) throws ContradictionException* where the filtering algorithm will be applied,
- *public ESat isEntailed()* where the entailment/satisfaction of the propagator is checked.

We now describe how these two methods can be implemented, plus an optional yet important method and the constructor parametrization.

Entailment

For debugging purpose or to enable constraint reification, a method named *isEntailed()* has to be implemented. The former is mainly used when implementing the constraint to make sure that found solutions respect the constraint specifications. The latter is called to valuate the boolean variable attached to a propagator when it is reified. The method returns *ESat.TRUE*, *ESat.FALSE* or *ESat.UNDEFINED* when respectively with respect to the current domain of the variables, the propagator can always be satisfied however they are instantiated, the propagator can never be satisfied and nothing can be deduced.

For example, consider the constraint $c = (x_1 + x_2 \leq 10)$ and the three following states:

- $x_1 = [1, 2], x_2 = [1, 2]$: the method returns *ESat.TRUE* since all combinations satisfy c,
- $x_1 = [22, 23], x_2 = [10, 12]$: the method returns *ESat.FALSE* since no combination satisfies c and
- $x_1 = [1, 10], x_2 = [1, 10]$: the method returns *ESat.UNDEFINED* since some combinations satisfy c, other don't.

Note: When an instance of a propagator is created, an array of its variables is automatically created and named *vars*. The order of elements of 'vars' shouldn't be modified: each variable knows its position in each of its propagators, modifying a position is only made by the solver itself.

The entailment method can be implemented as is:

```
1  Override
2  public ESat isEntailed() {
3      int sumUB = 0, sumLB = 0;
4      for (int i = 0; i < vars.length; i++) {
5          sumLB += vars[i].getLB();
6          sumUB += vars[i].getUB();
7      }
8      if (sumUB <= b) {
9          return ESat.TRUE;
10     }
11     if (sumLB > b) {
12         return ESat.FALSE;
13     }
14     return ESat.UNDEFINED;
15 }
```

Filtering algorithm

A propagator's first objective is to remove, from its variables domain, values that cannot belong to any solutions. This is the role of the *propagate(int m)* method. This method bases its deductions on the current domain of the variables and can update their domain on the fly. The expected state of this method exit is called a 'fix-point'.

Note: A local fix-point (wrt to a propagator) is reached when no more deductions can be done by a propagator on its variables. A global fix point ((wrt to a model) is reached when no more deductions can be done by any propagator on all variables.

Indeed, a propagator 'p' is not notified of its modifications but only those triggered by other propagators which modified at least one variable of 'p'. Each time one, at least, of its variable is modified, the satisfaction of a propagator need to check along with some filtering, if any, based on earlier modification.

Applying filtering rules can lead to a contradiction. In that case, the solver resumes after the filtering algorithm is stopped and manages to undo domain modification. Since restoring previous states is managed by the solver, it can safely be ignored when creating a propagator.

In the case of the sum constraint, *F* is computed first, then fast check of *F* is made to check obvious unsatisfaction and eventually a loop is operated over the variables to make sure that each upper bound is correct wrt to *F*. A simple loop is enough since *F* is computed reading $\overline{x_i}$ and writing $\underline{x_i}$.

Note that the method can throw an exception. An exception denotes that a failure is detected and the execution has to be stopped. In our case, if *F < 0* an exception should be thrown. In other cases, the methods that modify the variables domain can thrown such an exception too, when for example, the domain becomes empty.

The filtering method can be implemented as is:

```
1  @Override
2  public void propagate(int evtmask) throws ContradictionException {
3      int sumLB = 0;
4      for (int i  = 0; i < vars.length; i++) {
5          sumLB += vars[i].getLB();
6      }
7      int F = b - sumLB;
8      if (F < 0) {
9          fails();
10     }
11     for (int i  = 0; i < vars.length; i++) {
12         int lb = vars[i].getLB();
13         int ub = vars[i].getUB();
14         if (ub - lb > F) {
15             vars[i].updateUpperBound(F + lb, this);
16         }
17     }
18 }
```

The parameter of the method is ignored for now. On line 9, since the condition of unsatisfaction is met, a *ContradictionException* is thrown by calling *fails()*. On line 16, the i^{th} variable upper bound is updated. If the new value is greater or equal to than the current upper bound of the variable, nothing happens. If not, the variable is modified. If the new upper bound is lesser than the current lower bound, a *ContradictionException* is thrown automatically. Otherwise, the old upper bound is stored (for future restoration), the new upper bound is set and the propagators' list of the variable is iterated to inform each of them (except the one that triggers the event) that the variable domain has changed which can question their local fix-point.

Important: An *IntVar* can be modified in many ways: instantiation, upper bound modification, lower bound modification or value removal(s). These modifications can be achieved calling : *instantiateTo*,'updateUpperBound', *updateLowerBound*, *removeValue*, *removeValues*, , ...

Note: Some events can be *promoted*. For instance, when the new upper bound of a variable becomes equal to its current lower bound, the upper bound modification is promoted to an instantiation. The same goes with the new lower bound being equal to the current upper bound. Or when a value removal affects one bound, it is promoted to a bound modification (which in turn can be promoted to instantiation).

When the term 'value removal' is used it qualifies a hole in the middle of a variable domain, otherwise, due to

promotion, the most accurate term is used.

Propagation conditions (optional)

When a variables is modified, the type of *event* the modification corresponds is declared. For example when the upper bound of a variables is decreased, the event indicates *DEC_UPP*.

Not all types of event is relevant for all propagators and each of them can give its filtering conditions. By default, a propagator is informed of all type of modifications.

In our case, nothing can be done on value removal nor on upper bound modification. Thus, the following method can be override (note that is optional but leads to better performances):

```
@Override
public int getPropagationConditions(int vIdx) {
    return IntEventType.combine(IntEventType.INSTANTIATE, IntEventType.INCLOW);
}
```

Note that this method is called statically on each of its variables (denoted by *vIdx*) when posting the constraint to the model. Some propagators can thus declare distinct propagation conditions for each variable.

Constructor

Finally, any propagator should extends *Propagator* which is an abstract class and a call to *super* is expected as first instruction of the constructor.

Propagator abstract class provides three constructors but we will only depict one, the most important: *Propagator(V[] vars, PropagatorPriority priority, boolean reactToFineEvt)*.

The first argument is the list of variables, here an array of *IntVar*. The list of all variables the propagator can react on should be passed here. Consider that, with few exceptions, all variables of the propagator are expected.

The second parameter considers the filtering algorithm arity or complexity. There are seven ordered levels of priority, the three first ones (arity levels) are *UNARY*, *BINARY* and *TERNARY*. The three following ones (complexity levels) are *LINEAR*, *QUADRATIC*, *CUBIC*. Actually a *TERNARY* priority propagator is expected to run faster than a *QUADRATIC* priority one. So, considering the complexity instead of the arity may be more relevant when the filtering algorithm is very costly even if the propagator relies on only three variables.

The third parameter indicates if the propagator is able to react on fine events. This parameter will be presented in more details later on.

In our case, the input parameters are the array of *IntVar* 'x', the priority is based on the complexity which is linear in the number of variables and *false*. In addition, the constant 'b' needs to be stored too.

```
/**
 * Constructor of the specific sum propagator : x1 + x2 + ... + xn <= b
 * @param x array of integer variables
 * @param b a constant
 */
public MyPropagator(IntVar[] x, int b) {
    super(x, PropagatorPriority.LINEAR, false);
    this.b = b;
}
```

MyPropagator

A basic yet sound propagator which ensures that the sum of all variables is less than or equal to a constant is declared below.

```java
public class MyPropagator extends Propagator<IntVar> {

/**
 * The constant the sum cannot be greater than
 */
final int b;

/**
 * Constructor of the specific sum propagator : x1 + x2 + ... + xn <= b
 * @param x array of integer variables
 * @param b a constant
 */
public MyPropagator(IntVar[] x, int b) {
    super(x, PropagatorPriority.LINEAR, false);
    this.b = b;
}

@Override
public int getPropagationConditions(int vIdx) {
    return IntEventType.combine(IntEventType.INSTANTIATE, IntEventType.INCLOW);
}

@Override
public void propagate(int evtmask) throws ContradictionException {
    int sumLB = 0;
    for (IntVar var : vars) {
        sumLB += var.getLB();
    }
    int F = b - sumLB;
    if (F < 0) {
        fails();
    }
    for (IntVar var : vars) {
        int lb = var.getLB();
        int ub = var.getUB();
        if (ub - lb > F) {
            var.updateUpperBound(F + lb, this);
        }
    }
}

@Override
public ESat isEntailed() {
    int sumUB = 0, sumLB = 0;
    for (IntVar var : vars) {
        sumLB += var.getLB();
        sumUB += var.getUB();
    }
    if (sumUB <= b) {
        return ESat.TRUE;
    }
    if (sumLB > b) {
        return ESat.FALSE;
```

```
    }
    return ESat.UNDEFINED;
}
```

}

This first implementation outlines key concepts a propagator required. The entailment method should not ignored since it is helpful (even essential) to check the correctness of the implementation. The optional one which describes the propagation conditions can sometimes reduce the number of times a propagator is called without deducing new information (domain modifications or failure).

A more complex version

Based on Bounds Consistency Techniques for Long Linear Constraint, the first version can be improved in some ways.

We will consider first to desactivate the propagator when some conditions are satisfied, then we will show how backtrackable structures can be used and finally how a propagator can react to fine events.

Reduce to silence

An interesting feature available by default is the capacity to set passive a propagator that is entailed (i.e., is always true). Indeed, if all variables domain are in such state that any combinations satisfy the constraint, the propagator can be ignored in the propagation loop since it will not filter values nor fail.

In our case, this happens when the sum of the upper bounds is equal to or less than 'b'. If so, the propagator can safely be set to a passivate state in which it will not be informed of any new modifications occurring **in the current search sub-tree** (i.e., the propagator will be reactivated automatically on backtrack).

The filtering method can be modified like that:

```
@Override
public void propagate(int evtmask) throws ContradictionException {
    int sumLB = 0;
    for (int i  = 0; i < vars.length; i++) {
        sumLB += vars[i].getLB();
    }
    int F = b - sumLB;
    if (F < 0) {
        fails();
    }
    int sumUB = 0;
    for (int i  = 0; i < vars.length; i++) {
        int lb = vars[i].getLB();
        int ub = vars[i].getUB();
        if (ub - lb > F) {
            vars[i].updateUpperBound(F + lb, this);
        }
        sumUB += vars[i].getUB();
    }
    int E = sumUB - b;
    if (E <= 0) {
        this.setPassive();
    }
}
```

Line 18, a counter is updated with the sharpest upper bound of each variables. Line 21-23, if the condition is satisfied, the propagator is entailed and set to a passive state.

Note: We could also consider updating the propagation conditions to integrate upper bound modifications. Doing so, when one variable upper bound is modified, the entailment condition could be checked earlier.

Incrementally updating F

One may have noted that *F* is always computed as first step of *propagate(int evtmask)* method. On cases where few bounds are updated, there could be a benefit to incrementally compute *F*.

To compute *F* in an incremental way, three steps are needed: 1. creating a *backtrackable* int to record *F* but also variables' lower bound 2. initializing it on *propagate(int evtmask)* first call 3. anytime a variable is being modified, maintaining *F*

First, a *IStateInt* object and an *IStateInt* array are declared as class variables. In the propagator's constructor, through the *Model*, the objects are initialized:

```
1  /**
2   * The constant the sum cannot be greater than
3   */
4  final int b;
5
6  /**
7   * object to store F in an incremental way.
8   * Corresponds to a backtrackable int.
9   */
10 final IStateInt F;
11
12 /**
13  * array to store variables' previous lower bound.
14  * each cell is a backtrackable int.
15  */
16 final IStateInt[] prev_lbs;
17
18 /**
19  * Constructor of the specific sum propagator : x1 + x2 + ... + xn <= b
20  * @param x array of integer variables
21  * @param b a constant
22  */
23 public MyPropagator(IntVar[] x, int b) {
24     super(x, PropagatorPriority.LINEAR, false);
25     this.b = b;
26     this.F = this.model.getEnvironment().makeInt(0);
27     this.prev_lbs = new IStateInt[x.length];
28     for(int i = 0 ; i < x.length; i++){
29         prev_lbs[i] = this.model.getEnvironment().makeInt(0);
30     }
31 }
```

F is created with value *0*; its true value will be set on the first call to *propagate(int evtmask)* method. Same goes with *prev_ubs*. Any backtrackable primitive or operation is created thanks to the *environment* attached to the model. This ensures the integrity of the structure when backtracks occur.

The role of *prev_ubs* is to store the value of each variable lower bound. Then, anytime a variable lower bound is modified, its value can be retrieved and substracted from the current value to update *F*.

Second, *F* is initialized in the first call to *propagate(int evtmask)* method. This is where the value of *evtmask* is helpful. It can take 2 distinct values: one is dedicated to a full propagation, the other to a custom propagation. A full propagation is run on the initial propagation call, when each propagator is awaken by the solver. Then, if the propagator was declared not reacting to fine events (last parameter of the super constructor), full propagation is always run. On the other hand, if the propagator reacts to fine events, which will be the case for now, the initial propagation is kept full but then the main entry point of the filtering algorithm will be *propagate(int vIdx, int evtmask)* method (with two arguments). This method reacts to fine events, that means all variables modifications will be given as input thanks to the variable's index in *vars* (*vIdx*) and the event mask which is can be a combination of event types, like in propagation conditions.

Most of the time, this method is decomposed into a fast but naive filtering algorithm and a delayed call to a custom, presumably not fast, filtering algorithm. But it can be made of no filtering at all (that's the case here) or no delayed call to custom filtering algorithm.

In our case, we will only incrementally maintain *F* and then delegate the filtering to the custom propagation.

```
private void prepare(){
    int sumLB = 0;
    for(int i = 0 ; i < vars.length; i++){
        sumLB += vars[i].getLB();
        // set the current lower bound in 'prev_lbs'
        prev_lbs[i].set(vars[i].getLB());
    }
    // set the value of F
    F.set(b - sumLB);
}

@Override
public void propagate(int vIdx, int mask) throws ContradictionException {
    // 1. get the current lower bound of the modified variable
    int lb = vars[vIdx].getLB();
    // 2. update F with the difference between old and new lower bound
    F.add(lb - prev_lbs[vIdx].get());
    // 3. set the new lower bound
    prev_lbs[vIdx].set(lb);
    // 4. delegate the filtering later on
    forcePropagate(PropagatorEventType.CUSTOM_PROPAGATION);
}

@Override
public void propagate(int evtmask) throws ContradictionException {
    if(PropagatorEventType.isFullPropagation(evtmask)){
        // First call to the filtering algorithm, F is not up-to-date
        // so prepare initialize its value and 'prev_lbs'
        prepare();
    }
    if (F.get() < 0) {
        fails();
    }
    for (IntVar var : vars) {
        int lb = var.getLB();
        int ub = var.getUB();
        if (ub - lb > F.get()) {
            var.updateUpperBound(F.get() + lb, this);
        }
    }
}
```

A call to *forcePropagate(int evtmask)* will call *propagate(int evtmask)* only when all fine events are received. This

ensures that F is set to the correct value before filtering forbidden values.