Choco

Table of contents

1. Welcome

1.1. Welcome to Choco

1.1.1. Welcome !

This site is dedicated to the Choco constraint programming system. Choco is a java library for constraint satisfaction problems (CSP), constraint programming (CP) and explanation-based constraint solving (e-CP). It is built on a event-based propagation mechanism with backtrackable structures.

It is based on a previous version of Choco implemented in Claire and Palm, a Choco extension providing explanation tools.

You can find links to the last release on this page (http://choco.sourceforge.net/download.html) .

If you need more information or ask any questions about Choco, please use the following links:

- Forums are available for technical questions or to open discussions (features needed and so on)
- Lists are available to participate to the Choco extension: contributions or documentation for instance

Choco is an open-source software, distributed under a BSD licence (http://choco.sourceforge.net/licence.html) and hosted by sourceforge.net.

1.2. Choco Licence

1.3. Installation notes

Choco is a Java library distributed as Java archive (a `jar` file). To install it, you need to:

- download the Choco jar file here;
- modify the `CLASSPATH` environment variable.

1.3.1. Download of the Choco archive

The archive can be downloaded in the Choco project page in the SourceForge site, or more simply here. Put this archive in a directory of your hard disk (for instance `C:\Choco\`).

To use it easily, this `jar` file must be added to the `CLASSPATH` environment variable. The next section explains how to do that with Windows and UNIX environments.

1.3.2. The CLASSPATH environment variable

1.3.2.1. With Windows XP

- Right-click *My Computer*, and then click *Properties*.
- Click the *Advanced* tab.
- Click *Environment variables*.
- Search the `CLASSPATH` variable, if it does not exist create it with *New*.
- Edit its value with *Edit* and add for instance `C:\Choco\choco-x_xxx.jar`.

1.3.2.2. With UNIX-like environments

- Edit your shell config file (`.bashrc` for bash for instance).
- Add the following command: `export CLASSPATH=$CLASSPATH:/path/to/jar/file` or `setenv CLASSPATH $CLASSPATH:/path/to/jar/file` depending on your shell (typically `export` for bash and `setenv` for csh)

1.4. First Steps with Choco

Choco is provided as a Java archive (`jar` file). It can be used as soon as this archive is contained in the `CLASSPATH` variable of your environment (Refer to the installation page). We now suppose that the installation step has been correctly achieved.

1.4.1. A first example: Magic Square

The problem consists in filling a `NxN` grid with the numbers 1, 2, . . . , N such that each row and column has the same sum.

Let's create a new class `MagicSquare` in a file `MagicSquare.java`:

```
import choco.integer.IntVar;
import choco.Problem;

public class MagicSquare {

  public static void main(String[] args) {
    int n = 4;
    System.out.println("Magic Square Problem with n = " + n);
  }
}
```

We will now complete this main method.

1.4.1.1. Create the problem

First a new problem must be created. Creation of variables and constraints are based on the Problem object. Therefore, it is a central element of your choco program.

```
    Problem myPb = new Problem();
```

1.4.1.2. Create variables

Variables have to be created for this problem. As we deal with discrete domains to fill the magic grid, `EnumIntVar` variables will be used. Bounds Variables called `BoundIntVar` exist and can be used for larger domains. A table of size `NxN` is created here and a variable is associated to each cell of the grid. The name of the variable, the lower and upper bounds of its enumerated domain are given in argument of the variable constructor. Notice that one more variable, `sum`, indicating the sum of lines and rows will be needed.

```
    IntVar[] vars = new IntVar[n * n];
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) {
        vars[i * n + j] = myPb.makeEnumIntVar("C" + i + "_" + j, 1, n *
n);
      }
    IntVar sum = myPb.makeEnumIntVar("S", 1, n * n * (n * n + 1) / 2);
```

1.4.1.3. Add constraints

Again, the `Problem` instance is responsible for creating and posting the constraints: Firstly, all the cells of the grid have to take different values. Our variables should therefore be pairwise distinct :

```
    for (int i = 0; i < n * n; i++)
      for (int j = 0; j < i; j++)
        myPb.post(myPb.neq(vars[i], vars[j]));
```

Secondly, the variable `sum` is linked to the variables of each lines and rows. A scalar product with coefficients equal to 1 is used to ensure that the sum of each line and row will be equal to the same value (notice that a sum constraint is available as a shorthand).

```
    int[] coeffs = new int[n];
    for (int i = 0; i < n; i++) {
      coeffs[i] = 1;
    }

    for (int i = 0; i < n; i++) {
      IntVar[] col = new IntVar[n];
      IntVar[] row = new IntVar[n];

      for (int j = 0; j < n; j++) {
        col[j] = vars[i * n + j];
        row[j] = vars[j * n + i];
      }

      myPb.post(myPb.eq(myPb.scalar(coeffs, row), sum));
      myPb.post(myPb.eq(myPb.scalar(coeffs, col), sum));
    }
```

Finally, the value of the sum can be specified and should be equal to `N*(N*N + 1) / 2`. This constraint is a redundant one that ensure a stronger propagation. It can be omitted.

```
    myPb.post(myPb.eq(sum, n * (n*n + 1) / 2));
```

We can now either search for one solution, or all solutions of the problem.

1.4.1.4. Searching one solution

The first solution can be obtained with a simple call to the `solve()` method. It will give the first solution found by the solver and stop the search.

```
    myPb.solve();
```

1.4.1.5. Searching all solutions

To search all solutions, the `solveAll()` method can be called:

```
    problem.solveAll();
```

We can now use the solver instance to know how many solutions were found. For instance, let's display the number of solutions:

```
    System.out.println("Solution number: " +
      problem.getSolver().getNbSolutions());
```

Let's see what our choco program looks like :

```
import choco.integer.IntVar;
import choco.Problem;
```

```
public class MagicSquare {

  public static void main(String[] args) {
    int n = 4;
    System.out.println("Magic Square Problem with n = " + n);

    Problem myPb = new Problem();

    IntVar[] vars = new IntVar[n * n];
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) {
        vars[i * n + j] = myPb.makeEnumIntVar("C" + i + "_" + j, 1, n *
n);
      }
    IntVar sum = myPb.makeEnumIntVar("S", 1, n * n * (n * n + 1) / 2);

    myPb.post(myPb.eq(sum, n * (n*n + 1) / 2));
    for (int i = 0; i < n * n; i++)
      for (int j = 0; j < i; j++)
        myPb.post(myPb.neq(vars[i], vars[j]));

    int[] coeffs = new int[n];
    for (int i = 0; i < n; i++) {
      coeffs[i] = 1;
    }

    for (int i = 0; i < n; i++) {
      IntVar[] col = new IntVar[n];
      IntVar[] row = new IntVar[n];

      for (int j = 0; j < n; j++) {
        col[j] = vars[i * n + j];
        row[j] = vars[j * n + i];
      }

      myPb.post(myPb.eq(myPb.scalar(coeffs, row), sum));
      myPb.post(myPb.eq(myPb.scalar(coeffs, col), sum));
    }
    myPb.solve();
  }

}
```

1.4.2. Execution

Like all Java programs, executing this code needs to compile and launch it. If your `CLASSPATH` variable points to the Choco jar file, the following command allows to compile this sample class: `javac MagicSquare.java`.

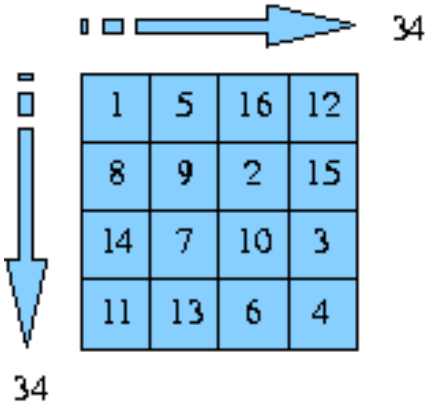
If your `CLASSPATH` does not contain the Choco package, please read the page about installation or use the `-classpath` option: `javac -classpath $CLASSPATH;/path/to/jar/file MagicSquare.java` (with semi-columns with Windows environment and columns for UNIX like environements).

Last, to launch the main method, you only need to call: `java MagicSquare`. Again, if

your CLASSPATH is not set up, you can use the -cp option. Here is how the result should look like:

```
>java MagicSquare
Magic Square Problem with n = 4
Pb[17 vars, 129 cons]
C0_0{1}
C0_1{2}
C0_2{15}
C0_3{16}
C1_0{6}
C1_1{11}
C1_2{10}
C1_3{7}
C2_0{13}
C2_1{9}
C2_2{4}
C2_3{8}
C3_0{14}
C3_1{12}
C3_2{5}
C3_3{3}
S{34}

-- solve => 1 solutions
-- 172[+0] millis.
-- 9[+0] nodes
```



A solution of the 4*4 magic square

Each solution is displayed as soon as it is found. The solution found here corresponds to the square displayed in the previous figure. We could ask for all solutions and display the number of solutions. Last, the log information is displayed asynchronously; here the log indicates the solution number and the time and node number needed to find the first solution.

1.4.3. Using explanations

Choco contains an explanation based solver based on PaLM. Switchintg from choco to Palm only require to change the type of the problem object. Since the PaLM problem inherits from the Choco one, all problem modelling is similar. For instance, you can use

`mac-dbt` algorithm by adding this import `import choco.palm.*;` and by modifying the line creating the problem instance: `Problem myPb = new PalmProblem();`. The rest of the code remains unchanged.

Of course using explanations is more usefull when you want to know why a value is removed from a variable's domain. For instance, if one want to know why `var3` can not equal 1, the following lines can be added (see the complete code for the dependent imports):

```
    Explanation expl = ((PalmProblem)problem).makeExplanation();
    ((PalmIntVar)vars[3]).self_explain(PalmIntDomain.VAL, 1, expl);
    System.out.println("Why " + vars[3] + " != 1 : " + expl);
```

The first line creates an explanation instance which is an entity able to store information explaining an inference: a set of constraints. The second line update this explanation with the explanation of the missing value 1 in the variable domain of the cell 3. Last, this explanation is displayed.

The complet code is now:

```
import choco.Problem;
import choco.integer.IntVar;
import choco.palm.PalmProblem;
import choco.palm.integer.PalmIntVar;
import choco.palm.integer.PalmIntDomain;
import choco.palm.explain.Explanation;

public class MagicSquare {

  public static void main(String[] args) {
    int n = 4;
    System.out.println("Magic Square Problem with n = " + n);

    Problem myPb = new PalmProblem();

    IntVar[] vars = new IntVar[n * n];
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) {
        vars[i * n + j] = myPb.makeEnumIntVar("C" + i + "_" + j, 1, n *
n);
      }
    IntVar sum = myPb.makeEnumIntVar("S", 1, n * n * (n * n + 1) / 2);

    myPb.post(myPb.eq(sum, n * (n*n + 1) / 2));
    for (int i = 0; i < n * n; i++)
      for (int j = 0; j < i; j++)
        myPb.post(myPb.neq(vars[i], vars[j]));

    int[] coeffs = new int[n];
    for (int i = 0; i < n; i++) {
      coeffs[i] = 1;
    }

    for (int i = 0; i < n; i++) {
      IntVar[] col = new IntVar[n];
      IntVar[] row = new IntVar[n];
```

```
      for (int j = 0; j < n; j++) {
        col[j] = vars[i * n + j];
        row[j] = vars[j * n + i];
      }

      myPb.post(myPb.eq(myPb.scalar(coeffs, row), sum));
      myPb.post(myPb.eq(myPb.scalar(coeffs, col), sum));
    }

    myPb.solve();

    Explanation expl = ((PalmProblem)myPb).makeExplanation();
    ((PalmIntVar)vars[3]).self_explain(PalmIntDomain.VAL, 2, expl);
    System.out.println("Why " + vars[3] + " =! 2 : " + expl);

  }
```

The execution of this code should look like:

```
>java Sample
Magic Square Problem with n = 4
** JPaLM : Constraint Programming with Explanations
** JPaLM v0.1 (September, 2003), Copyright (c) 2000-2003 N. Jussien
Why C0_3 =! 1 : {C0_3 !== C0_0 + 0, C0_0 == 1}
```

This shows that `var3` can not equal to 1 because of two constraints:

- `C0_3 !== C0_1`
- `C0_0 == 1`

Notice that the second constraint is a decision taken during search by the solver. Indeed since `var0` is equal to 1 and `var3` has to be distinct of `var0`, it can not be equal to 1.

2. Download

2.1. Choco downloads

2.1.1. Latest release

This section is intended for most users. You will find here the latest stable version published. Included with the release come a documentation, code samples, and more.

- The jar archive of the choco java package
- A zip archive of choco java sources
- A zip archive of the (javadoc) API
- A zip archive of code samples

2.1.2. Unstable version

This version contains last modifications, but was not completely tested and validated.

- The jar archive of the choco java package
- A zip archive of choco java sources
- A zip archive of the (javadoc) API

3. Community

3.1. History of Changes

RSS

3.1.1. Version 0_9b003 (22/09/2004)

- Stuff to implement incremental solve ()

3.1.2. Version 0_9b002 (17/06/2004)

- Set variables and constraints (hca)
- Real variables and constraints (gro)
- Some new examples ()
- Solve API: solve() and solveAll() (fla)

3.1.3. Version 0_9b001 (11/05/2004)

- Initial version (fla)

3.2. Frequently Asked Questions

3.2.1. Questions

3.2.1.1. 1. Choco installation

1.1. How to install Choco ?

Choco is provided as a `jar` file. You only need to had this file to your `CLASSPATH`.

3.3. Choco Performance Report

3.3.1. Benchmark Description

The benchmarck is done on a Pentium 4 3Gh, 512 Mo with the sun jdk1.4.2_03 in server mode (-server). The stack size used is the default jvm size (no -X options).

Note:

By default, the mindom heuristic for choosing variable and the increasing domain heuristic for choosing value are used. Ties between variables are broken with the order where variables were created (The first minimum domain variable created is selected).

3.3.1.1. N Queens

The problem is modeled with N integer variables of domain 1 .. N corresponding to each

line of the board, and only inequality constraints are used.

3.3.1.2. Magic Square

The problem is modeled with N * N integer variables of domain 1 .. N * N corresponding to each cell of the square, and linear combinations are posted for each line, each column and the two diagonals (the magic number is given a priori).

3.3.1.3. Golomb Rules

N variable for all distances (all pairs of mark) are used. Differences between all distances are stated with a BoundAllDiff constraint. Moreover, a global redondant constraint is added. The variable selection is done in the order of distances (from 0 to 1,2,3, then from 1 to 2,3,4, etc ...) : the first non instantiated distance is choosen.

3.3.1.4. Latin Square

The problem is modeled with N * N integer variables of domain 1 .. N corresponding to each cell of the square, and uses simple inequalities between variables of the same line and column.

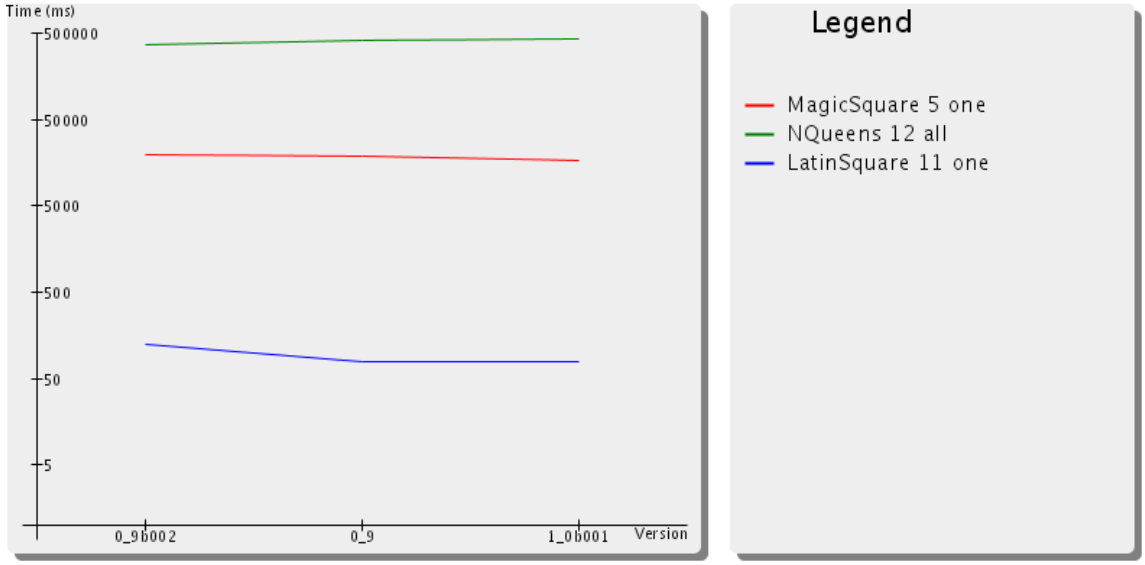
3.3.2. Last Version Performance (in ms)

Problems	MAC	CBJ	MACDBT	DR
MagicSquare 4 all	45641	241781	N/A	N/A
MagicSquare 5 one	688	1656	16578	6281
NQueens 12 all	11532	52235	427922	N/A
NQueens 13 all	59688	298015	N/A	N/A
NQueens 14 all	352704	N/A	N/A	N/A
GolombProblem 11 one	101078	N/A	N/A	N/A
GolombProblem 12 one	970484	N/A	N/A	N/A
LatinSquare 11 one	N/A	47	78	78

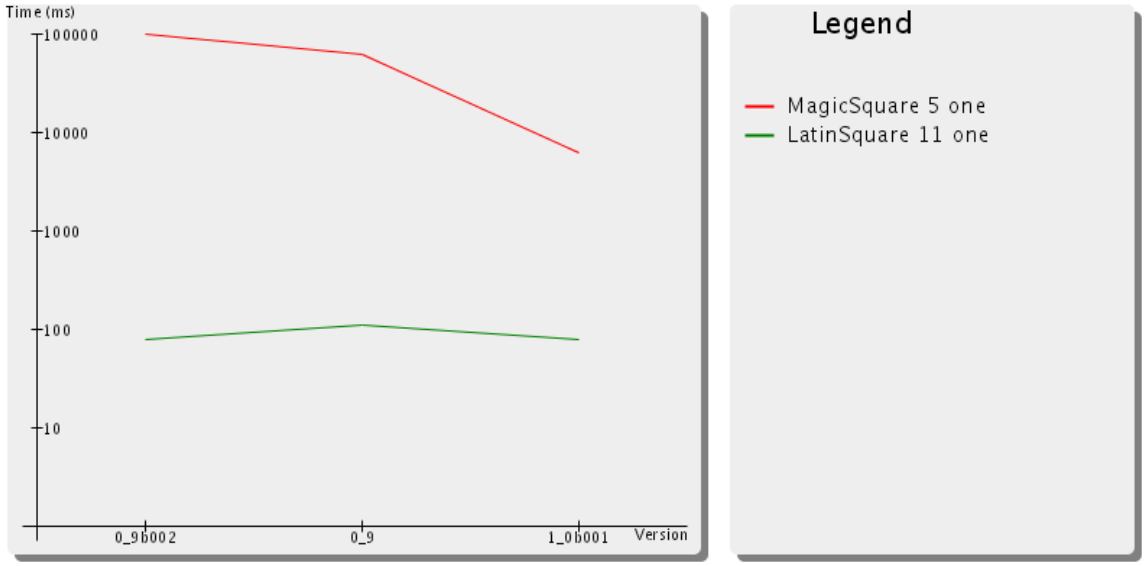
Table 1: Performance report on version '1_0b001'

3.3.3. Performance evolution

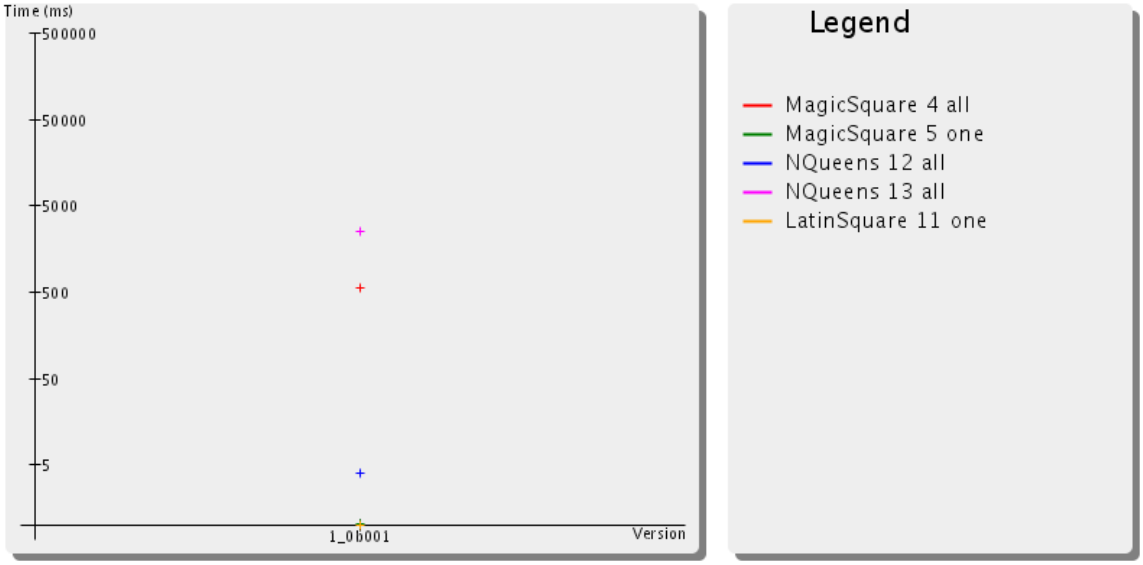
3.3.3.1. MACDBT



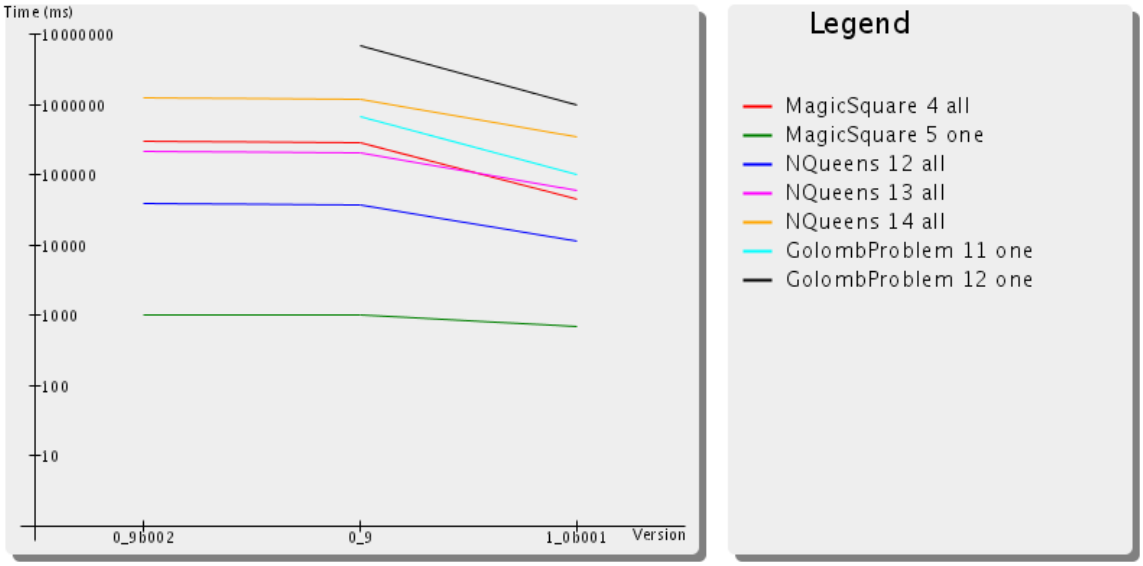
3.3.3.2. DR



3.3.3.3. CBJ



3.3.3.4. MAC



3.4. Choco contribution

3.4.1. How to contibute ?

If you want to contribute to the Choco development you can help our team:

- by providing some patches for bugs you find in Choco; in this case contact us on the mailing list choco-users;
- by providing some constraints we will put in the repository below; in this case send a

jar file and a brief description to Guillaume Rochart

3.4.2. Contribution constraints

Here you can find some contribution from Choco users: each jar file contains some constraints you can test and use even if they were not tested by the Choco team.

3.4.2.1. Bound All Different (Guillaume Rochart)

This constraints is a bound version of the allDifferent constraint which is particularly useful when you only want to deal with bound variables. This is a Choco port of the constraint proposed in the article of A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek.

Here you can find the bad.jar file.

4. Resources

4.1. User guide

4.1.1. Creating the Problem

The central element of a choco program is the `Problem` object.

```
Problem myPb = new Problem();
```

Switching from choco to palm only require to use a specific Problem : a PalmProblem.

```
Problem myPb = new PalmProblem();
```

4.1.1.1. Variables

Actually, Three main kind of variables exist :

- `IntVar` : It describes discrete domains where values are integers.
  - `EnumIntVar` : It corresponds to enumerated domains and should be used when discrete and quite small domains are needed.
  - `BoundIntVar` : Such domains are represented by their lower and upper bounds (propagation is only perform on the bounds). One can use them when large domains are needed.
- `RealVar` : It describes continous domain and use intervals to represent values.
- `SetVar` : It describes discrete set domains where a value of a variable is a set. Set vars are encoded with two classical bounds : the union of the all set of possible values called the envelope and the intersection of all set of possible values called the kernel.

Once the Problem has been created, variables are created through factories available on the Problem instead of the classical java constructor. The use of factories allows to

redefine them in a specific Problem (as done for the PalmProblem) and ensures that constraints and variables types will remain compatible. The following example of code show how to create a finite domain variable:

```
IntVar v1 = myPb.makeEnumIntVar("var1", 1, 10);
```

v1 is an enumerated variable which is called var1 and has a discrete domain from 1 to 10. It has been created for the problem myPb.

Integer Variables

1. `makeBoundIntVar(String s, int lb, int ub)` : creates a finite domain variable with domain (lb .. ub), with name s.
2. `makeEnumIntVar(String s, int lb, int ub)` : creates a finite domain variable whose domain is approximated by bounds (lb .. ub), with name s.

The state of an IntVar can be accessed through the main following public methods on the IntVar class:

1. `hasEnumeratedDomain()`: checks if the variable is an enumerated or a bound one.
2. `getInf()`: returns the lower bound of the variable.
3. `getSup()`: returns the upper bound of the variable.
4. `getValue()`: returns the value if it is instantiated.
5. `isInstantiated()`: checks if the domain is reduced to a singleton.
6. `canBeInstantiatedTo(int val)`: checks if the value val is contained in the domain of the variable.
7. `getDomainSize()`: returns the current size of the domain.

The domain of an IntVar can be modified through the main following public methods: Such operations are subject to the backtrack mechanism.

1. `setMin()`: set the lower bound of the variable.
2. `setMax()`: set the upper bound of the variable.
3. `setVal()`: set the value of the variable.

Set Variables

Set variables are still under development but a bit of api is still available.

1. `makeSetVar(String name, int l, int u)`: creates a set domain variable with name s where `l` correponds to the lower bound of the inital enveloppe and `b` the upper bound.

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

1. `isInDomainKernel(int x)`: checks if a value x is contained in the current kernel.
2. `isInDomainEnveloppe(int x)`: checks if a value x is contained in the current

envelope.
3. `getDomain()`: returns the domain of the variable as a `SetDomain`. Iterators on envelope or kernel can than be called.
4. `GetKernelDomainSize()`: returns the size of the kernel.
5. `GetEnveloppeDomainSize()`: returns the size of the kernel.
6. `GetEnveloppeInf()`: returns the first available value of the envelope.
7. `GetEnveloppeSup()`: returns the last available value of the envelope.
8. `GetKernelInf()`: returns the first available value of the kernel.
9. `GetKernelSup()`: returns the last available value of the kernel.
10. `getValue()` : returns a table of integer int[] containing the current domain.

The domain of a SetVar can be modified through the main following public methods(ContradictionException can be thrown in case of an inconsistent change) :

1. `setValIn(int x)`: set a value inside the kernel.
2. `setValOut(int x)`: set a value outside the kernel.
3. `setVal(int[] val)`: set the value of the variable.

Real Variables

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

1. `makeRealVar(String s, double lb, double ub)` : creates a continous domain variable whose domain is considered as an interval [lb,ub], with name s.
2. `getInf()`: returns the lower bound of the variable (a double).
3. `getSup()`: returns the upper bound of the variable (a double).
4. `isInstantiated()`: checks if the domain of a variable is reduced to a canonical interval. A canonical interval indicates that the domain has reached the precision given by the user or the solver.

4.1.1.2. Constraints

Constraints are represented by dedicated objects organized in a class hierarchy. It encapsulates a filtering algorithm which are called when a propagation step occur or when external events happen on the variables belonging to the constraint, such as value removals or bounds modification.

A constraint is stated to a problem by using the method post available on the Problem object : post(Constraint c). Creating a constraint and adding it to the constraint network can be done using the Problem api. For example, adding a constraint of difference between two variables is simply written as follow:

```
myPb.post(myPb.neq(vars1, vars2));
```

Integer constraints

Basic constraints

The constraints available with choco are arithmetic constraints (equality, difference, comparisons and linear combination), user-defined binary constraints (AC4,AC3, ...), boolean operators (or, and, implies, ...).

The simplest constraints are comparisons which are defined over expressions of variables such as linear combinations. The following comparison constraints can be accessed through the Problem API:

1. `neq(IntExp v1, IntExp v2)` : v1 != v2.
2. `eq(IntExp v1, IntExp v2)` : v1 = v2.
3. `leq(IntExp v1, IntExp v2)` : v1 <= v2.
4. `lt(IntExp v1, IntExp v2)` : v1 < v2.

To construct complex expressions of variables, simple operators can be used:

1. `minus(IntExp exp1, IntExp exp2)`: exp1 - exp2.
2. `plus(IntExp exp1, IntExp exp2)`: exp1 + exp2.
3. `mult(int coef, IntExp exp)`: coef * exp.
4. `scalar(int[] coef, IntVar[] vars)`: coef[1]*vars[1] + ... + coef[n]*vars[n].
5. `sum(IntVar[] vars)`: vars[1] + ... + vars[n].

User-defined constraints

Choco supports the statement of constraints defined by arbitrary relations. It offers the possibility of stating binary constraints with several AC Algorithm and also n-ary constraints with a weaker form of propagation.

Binary constraints

The relation defines feasible or infeasible pairs of values for the two variables involved in the constraint. Relations may be defined by two means:

1. Tables : specifying those pairs of value for which the constraints is satisfied/unsatisfied.
2. Predicates : specfying the method to be called in order to check whether a pair of value is feasible or not.

One the one hand constraints based on tables of values may become rather memory consuming in case of large domains, although relation tables may be shared by different constraints. On the other hand, predicate constraints require little memory as they do not cache thruth 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.

The creation of a constraint for a relation can be done through the following Problem API

:

1. makePairAC(IntVar v1, IntVar v2, ArrayList pairs, boolean feas, int ac).
2. makePairAC(IntVar v1, IntVar v2, boolean[][] pairs, boolean feas, int ac).
3. relationPairAC(IntVar v1, IntVar v2, BinRelation pairs, int ac).

Parameter feas indicates whether the relation models feasible pairs of values or infeasible one (default is infeasible). Parameters pairs contains the definition of the relation. A list of int[] of size 2 in the first case, a boolean[][] in the second case or as a BinRelation in the last case. Finally parameter ac selects the algorithm for enforcing arc-consistency (default ac = 2001). Supported values for this parameter are :

1. 3 for AC3 algorithm (searching from scratch for supports on all values)
2. 4 for AC4 algorithm (maintaining a count of supports for each value)
3. 2001 for the AC2001 algorithm (maintaining the current support of each value)

The definition of a binary relation based on a predicate can be done by inheriting from the CouplesTest class. Have a look on the following example :

```
public class MyInequality extends CouplesTest{

public boolean checkCouple(int x, int y) {

return x != y;

}

}
```

You can then state a constraint as the following :

```
pb.post(pb.relationPairAC(v1, v2,new MyInequality()));
```

The complete Problem API allow to easily create binary constraint :

1. infeasPairAC(IntVar v1, IntVar v2, ArrayList pairs)
2. feasPairAC(IntVar v1, IntVar v2, ArrayList pairs)
3. relationPairAC(IntVar v1, IntVar v2, BinRelation binR)
4. ...

N-Ary Constraints

The situations for binary constraints is extended to the case of relations involving more than two variables, upto a significant difference from the propagation point of view: The propagation engine maintains arc-consistency for binary constraints throughout the solving process, while for n-ary constraints, it uses a weaker propagation mechanism with a forward checking algorithm.

The API for creating such constraints is the following ones:

1. makeTupleFC(IntVar[] vs, ArrayList tuples, boolean feas)

2. relationTuple(IntVar[] vs, LargeRelation rela)

Defining a specific n-ary relation without storing the tuples can be done as on the following example :

```
public class NotAllEqual extends TuplesTest {


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


}
```

Otherwise, the tuples are given in an ArrayList as int[] table given the compatible/incompatible values. One can then state the constraint to the problem :

```
pb.post(pb.relationTuple(new IntVar[]{x, y, z}, new NotAllEqual()))
```

Finally, the structure of the Consistency relations can be seen in more details on the following picture :

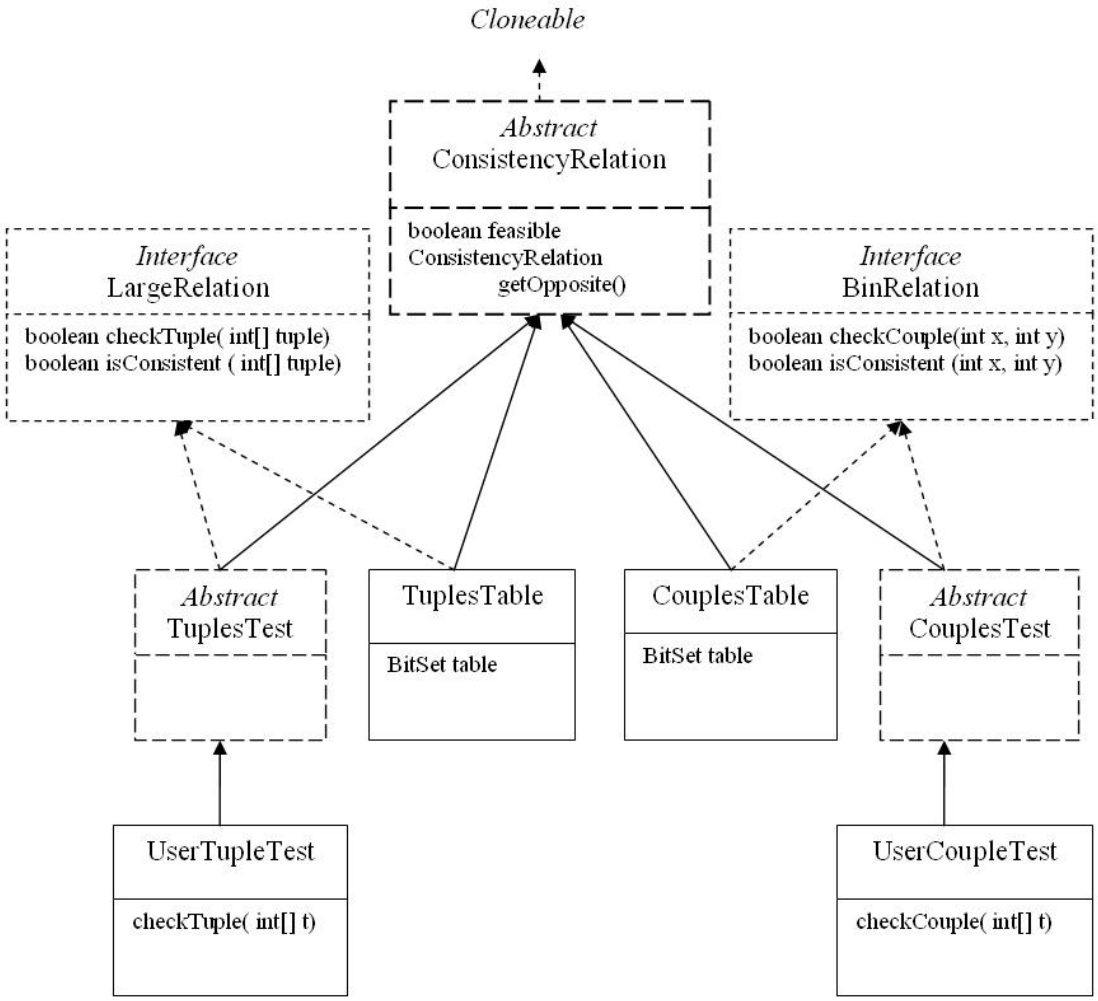


Image1

Advanced constraints

Choco includes several global constraints. Those constraints allows to filter efficiently some inconsistant values. For instance, if several variables should be affected to different values, using a global constraint can offer some additionnal filtering rules (for instance a should be in [1,4], b in [1,4], c in [3,4] and d in [3,4], then one can deduce that a and b cannot be instantiated to 3 or 4; such rule cannot be infered by simple binary constraints).

Here are described som of those constraints :

1. pb.allDifferent(IntVar[] vars) creates a constraint ensuring that all pairs of variable have distinct values (which is useful for some matching problems);
2. pb.occurrence(IntVar[] vars, int value, IntVar occurrence) creates a constraint to

ensure that occurrence will be instantiated to the number of occurrences of valu ein the list of variables vars; this is a specilisation of the following constraint;
3. pb.globalCardinality(IntVar[] vars, int[] low, int[] up) creates a constraint ensuring that the number of occurrences of the value 1 in all the variables vars is between low[0] and up[0], and generallay the number of occurrences of the value i in vars is between low[i-1] and up[i-1].

Some other global constraints can be added to Choco in future releases. One can find all the API and constraints available on the Javadoc API.

Set constraints

Choco supports the statement of constraints among sets accessible on the `SetModeler` class. The api for SetConstraint is available on the SetModeler as show on the following code example :

```
pb = new Problem();

SetModeler mod = new SetModeler(pb);

pb.post(mod.eqCard(vars[i],3));
```

The following set constraints are available :

1. `member(SetVar sv1, int val)`: states that the variable sv1 contains value val.
2. `notMember(SetVar sv1, int val)`: states that the variable sv1 does not contain value val.
3. `setDisjoint(SetVar sv1, SetVar sv2):` states that sv1 and sv2 are disjoint sets ; e.g. that sv1 and sv2 contain no common values.
4. `setInter(SetVar sv1, SetVar sv2, SetVar inter):` states that the `inter` set variable is the intersection of set variables sv1 and sv2 ; e.g. states that `inter` contains exactly those values contained in both sets sv1 and sv2.
5. `eqCard(SetVar sv, int val)`: states that the cardinality of the set variable is equal to value `val.`
6. `geqCard(SetVar sv, int val)`: states that the cardinality of the set variable is greater or equal equal than value `val`.
7. `leqCard(SetVar sv, int val)`: states that the cardinality of the set variable is equal than value `val`.

To deal with integer variables, the following mixed constraint are available :

1. `member(SetVar sv1, IntVar var).`
2. `notMember(SetVar sv1, IntVar var).`
3. `eqCard(SetVar sv, IntVar iv).`
4. `geqCard(SetVar sv, IntVar iv).`
5. `leqCard(SetVar sv, IntVar iv).`

Real constraints

4.1.2. Search and Branching

4.1.2.1. Search for one or all solutions

When the problem is modelled and created thanks to the API described in previous sections, one may want to solve it ! If only one solution is needed, this is quite easy. The following code solves the problem and display the solution :

```
if (pb.solve() == Boolean.TRUE) {
for(int i = 0; i < pb.getNbIntVars(); i ++) {
System.out.println(pb.getIntVar(i) + " = " +
pb.getIntVar(i).getValue());
}
}
```

If one wants several solutions, the incremental solve API can be used : `nextSolution` searches for another solution in the search tree :

```
if (pb.solve() == Boolean.TRUE) {
do {
for(int i = 0; i < pb.getNbIntVars(); i ++) {
System.out.println(pb.getIntVar(i) + " = " +
pb.getIntVar(i).getValue());
}
} while(pb.nextSolution() == Boolean.TRUE);
}
```

4.1.2.2. Optimization

Optimization is done in Choco according to a variable denoting the objective value. The objective function is then expressed as a constraint over this variable and the rest of the problem. The API concerning optimization proposes to minimize/maximize this objective variable (instead of a call to pb.solve()) :

1. minimize(IntVar obj, boolean restart).
2. maximize(IntVar obj, boolean restart).

Parameter restart is a boolean indicating whether the solver will restart the search after

each solution found or if it will keep backtracking from the leaf of the last solution found.

Look at the following knapsack example where a scalar product over three variables is maximized :

```
Problem pb = new Problem();

IntVar obj1 = pb.makeEnumIntVar("obj1",0,7);

IntVar obj2 = pb.makeEnumIntVar("obj1",0,5);

IntVar obj3 = pb.makeEnumIntVar("obj1",0,3);

IntVar cost = pb.makeBoundIntVar("cout",0,1000000);

int capacite = 34;

int[] volumes = new int[]{7,5,3};

int[] energie = new int[]{6,4,2};


// capacity constraint

pb.post(pb.leq(pb.scalar(volumes,new
IntVar[]{obj1,obj2,obj3}),capacite));


// objective function

pb.post(pb.eq(pb.scalar(energie,new IntVar[]{obj1,obj2,obj3}),cost));


pb.maximize(c,false);
```

4.1.2.3. Limiting the search space

Limits may be imposed on the search algorithm to avoid spending too much time in the exploration. The limits are updated and checked each time a new node is created . The API is given on the Solver class:

1. setTimeLimit(int timeLimit): stops the search algorithm after timeLimit milliseconds have been spent searching.
2. setNodeLimit(int nodeLimit): stops the search algorithm after nodeLimit nodes have been expanded.

For example, to stop the search after 30 seconds :

```
pb.getSolver().setTimeLimit(30000);
```

To define your own limits/statitistics, you can create a limit object by extending the AbstractGlobalSearchLimit class. Documentation about the definition of your own limits

will come later.

4.1.2.4. Define your own tree search

The construction of the search tree is done according to a serie of Branching objects (that plays the role of achieving intermediate goals in logic programming). The user may specify the sequence of branching objects to be used to build the search tree. For the moment, choco proposes an AssignVar branching that proceeds in assigning variables to values. More complex branching schemes will come soon.

Variable/value selection for AssignVar branching

Choco provides means of composing search trees by specifying the heuristics used for selecting variables and values on integer variables in case of AssignVar branchings. These primitives are based on the following interfaces :

1. IIntVarSelector : Interface for the variable selection
2. IValIterator : Interface that provide a way of describing an iteration scheme on the domain of a variable
3. IValSelector : Interface for a value selection

Default branching heuristics

The default branching heuristic used by Choco is to choose the variable with current minimum domain size first and to take its values in increasing order. The default branchings currently supported by choco are available in the packages choco.integer.search for integer variables (choco.set.search for set variables). Concrete examples of the previous interfaces are the classes MinDomain, MostConstrained, DomOverDeg, RandomIntVarSelector ...

To change the default branching, you can use the API available on the Solver class as shown on the following example :

```
pb.getSolver().setVarSelector(new RandomIntVarSelector(pb));
```

Changing the values enumeration/selection can be done in the same way:

```
// select the value in increasing order
pb.getSolver().setValIterator(new DecreasingDomain());
// or select a random value
pb.getSolver().setValSelector(new RandomIntValSelector());
```

How to define it own AssignVar branching

You may extend this small library of branching schemes and heuristics by defining your

own concrete classes of IIntVarSelector, IValIterator and IValSelector.

Let's take the example of a static variable ordering :

```
public class StaticVarOrder implements IIntVarSelector {
/**
* the sequence of variables that need be instantiated
*/
protected IntVar[] vars;

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

public IntVar 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 the method selectIntVar which belongs to the contract of IIntVarSelector. Once the branching is finisched, it returns null and the next branching (if one exist) is taken by the search algorithm to continue the search, otherwise, the search stops as all variable are instantiated.

4.1.3. How to create its own constraint

4.1.3.1. The constraint hierarchy

A constraint must respect the interface `Constraint` to be handled by Choco. As a lot of low level methods (to manage the constraint network using a system of listeners) are

included in this interface, an abstract class `AbstractConstraint` provide an implementation of all listeners.

For integer variables, the easiest way to implement your own constraint is to inherit from one of the following classes :

- `AbstractUnIntConstraint`, `AbstractBinIntConstraint`, `AbstractTernIntConstraint` : A default implementation for constraint stating one, two or three variables.
- `AbstractLargeIntConstraint` : A default implementation for constraint stating more than 3 variables.

In the same way, SetConstraint can inherit from :

- `AbstractUnSetConstraint`, `AbstractBinSetConstraint`, `AbstractTernSetConstraint.`
- `AbstractLargeSetConstraint.`

Moreover, Constraints stating on integer and set variables can be written by inheriting from `AbstractMixedConstraint.`

4.1.3.2. Backtrackable structures

4.1.3.3. Updating domains of variable (indexes and links with propagation)

4.1.3.4. The event system handled by the propagation mechanism (constawake, asynchronous)

4.2. Samples

4.2.1. Choco samples

To see these samples, you need the Java Plugin (version >= 1.4)

Your browser is completely ignoring the <APPLET> tag!

5. Tutorials

5.1. tut_base

5.1.1. Tutorial : first steps with Choco

This tutorial is aimed at presenting the basic information needed to be able to write one's first application with Choco :

- • how to create a problem,

- how to solve it,
- how to explore the set of solution(s).

Our running example will be the n-queens problem.

Problem modeling

In order to solve the n-queens problem with Choco, the first thing to do is to create a problem. Here is how it is done in Choco:

```
Problem nQueens = new Problem();
```

This object will be the basis for all the subsequent information that will be added. A classical model for the n-queens problem consists in defining n variables taking their value between `1` and `n` : those variables represent the position of each queen on each line. In Choco, we can therefore declare an array of variables and ask to the `Problem` to initialise them:

```
IntVar[] queens = new IntVar[n]; // declaring an array of n variables
for (int i = 0; i < queens.length; i++) {
queens[i] = nQueens.makeEnumIntVar("Queen" + i, 1, n);
}
```

Note:

We used here `makeEnumIntVar` to declare an enumerated variable because we want to explicitely handle holes in the domain of the variable. Other possibilities exist: makeBoundIntVar, etc.

The next step consists in adding constraints to our problem:

- any two queens should be positionned in different lines (the model itself enforces this constraint)
- any two queens should be in different columns
- any two queens should not be in the same diagonal

We will use difference and linear combination constraints:

```
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
nQueens.post(nQueens.neq(queens[i], queens[j])); // different columns
nQueens.post(nQueens.neq(queens[i],
```

nQueens.plus(queens[j], j – i))); // diagonal

nQueens.post(nQueens.neq(queens[i],

nQueens.minus(queens[j], j – i))); // diagonal

```
}
}
```

Our problem is now completed: the problem has been created, variables have been created, constraints have been created and posted (they are now active in the problem). Now, it is time to actually solve the problem.

Solving the problem

The `Problem` object provides three methods that can be used to solve a problem: `propagate`, `solve`, and `solveAll`.

The first one can be used to propagate the constraints of the problem i.e. as much filtering as possible impossible values from the domain of the variables (no enumeration is performed here).

```
try {
nQueens.propagate();
}
catch (ContradictionException e) {
System.err.println("This problem is over-constrained");
System.exit(-1);
}
```

Warning:

As the `propagate` method is only in charge of propagating the constraints of the problem, it may happen that a contradaction is raised. This is why any call to this method should be embedded within a `try` / `catch` environment.

The second one is used to really solve the problem i.e. looking for one solution. It returns a Boolean:

- Boolean.TRUE if the problem does have a solution
- `Boolean.FALSE` if it does not

```
if (nQueens.solve() == Boolean.TRUE) {
System.out.println("the problem has at least one solution");
}
else {
```

```
System.out.println("the problem has no solution");
}
```

Warning:
The `solve` method may return `null` if the search has been interrupted because of a time or backtrack limit set up for the considered problem.

FIXME ():
Write a tutorial about search to be referred to in the previous warning.

The third one is used to compute all the solutions of the problem.

```
nQueens.solveAll();
```
System.out.println("the problem has " + nQueens.getSolver().getNbSolutions()

+ " solutions");

Exploring the set of solutions

Choco allows the user to print or explore the set of solutions. Using the `solve` method leaves the variables with the values of the computed solution if it exists.

```
nQueens.solve();
System.out.println("One solution is:");
for (int i = 0; i < queens.length; i++) {
System.out.println(queens[i] + " = " + queens[i].getValue());
}
```

Note:
Another way of accessing the variables of the problem is to use the following code:

```
for (int = 0; i < nQueens.getNbIntVars(); i++) {
System.out.println(nQueens.getIntVar(i) + " = "
+ nQueens.getIntVar(i).getValue());
}
```

Choco can also be used to enumerate the set of solutions for a problem. The first step is to `solve` the problem, and then to enumerate the solutions using the `nextSolution()` method.

```
if (nQueens.solve() == Boolean.TRUE) {
do {
for (int i = 0; i < nQueens.getNbIntVars(); i++) {
Systemm.out.println(nQueens.getIntVar(i) + " = "
+ pb.getIntVar(i).getValue());
}
} while (pb.nextSolution() == Boolean.TRUE)
}
```

Finally, at any time the `nQueens.getSolver().getSearchSolver().maxNbSolutionStored` (5 by default) solutions are accessible in the `nQueens.getSolver().getSearchSolver().solution` array (the cell number 0 containing the last computed one).

```

System.out.println("the current solution is: ");
for (int = 0; i < nQueens.getNbIntVars(); i++) {
System.out.println(nQueens.getIntVar(i) + " = "
+ nQueens.getIntVar(i).getValue());
}

Solution previousSolution =
nQueens.pb.getSolver().getSearchSolver().solutions;

System.out.println("the previsous solution was: ");
for (int = 0; i < nQueens.getNbIntVars(); i++) {
System.out.println(nQueens.getIntVar(i) + " = "
+ previousSolution.getValue(i));
```

```
}
```

At the end of this tutorial, we are now able to model, solve and explore the set of solutions on the n-queens problem.

5.2. Tutoriel : Créer sa propre contrainte globale

Le but de ce tutoriel est de présenter le fonctionnement interne d'une contrainte globale et son implémentation. Comme illustration, nous prendrons la contrainte `occur` qui permet de contraindre le nombre d'apparitions d'une valeur dans un ensemble de variales.

5.2.1. Sémantique de la contrainte

Nous commençons dans cette partie par préciser la sémantique souhaitée pour cette contrainte. Cette contrainte a pour paramètres :

- un ensemle de variables (x1...xn) dont une variable particulière xn notée aussi nbOcc;
- une valeur étudiée v : il s'agit de la valeur dont on contraint le nombre d'occurences;
- deux booléens spécifiant si l'on contraint le nombre d'ocurrences minimal, maximal, ou encore les deux.

Si l'on souhaite contraindre le nombre minimal d'occurences de la valeur v parmis x1...xn-1, il faut vérifier à chaque moment qu'il y a toujours suffisamment d'occurences de v dans les valeurs *possibles* des domaines associés aux variables. De même, si l'on souhaite contraindre le nombre maximal d'occurences, il faut vérifier le nombre d'occurences *nécessaires*, c'est-à-dire le nombre de variables déjà instanciées à la valeur v. Dans le cas où l'on souhaite contraindre les deux bornes, il faut utiilser ces deux raisonnments.

Par exemple :

- si Dx1={1,2,4}, Dx2={2,3}, Dx3={1,4} et DnbOcc={1,3}, et que l'on cherche à contraindre le nombre d'occurences minimal de la valeur v=3, il est claire qu'il ne reste qu'une seule variable pouvant être affectée à v, on peut donc en inférer x2=3.
- si maintenant Dx1={3}, Dx2={1,3,4}, Dx3={3} et DnbOcc={1,2), et que l'on cherche à contraindre le nombre d'occurences maximal de la valeur v=3, on peut inférer que x2#3.

5.2.2. Implémentation d'une contrainte sans explication

Pour simplifier, nous commençons par décrire l'implémentation de la contrainte `occur` dans le cas non expliqué. Comme nous le verrons, ajouter des explications dans des contraintes globales peut demander du travail supplémentaire. S'agissant d'une contrainte prenant en compte de nombreuses variables, on peut créer la classe en étendant `AbstractLargeIntConstraint` qui permet d'implémenter de telles contraintes.

```
public class Occurence extends AbstractLargeIntConstraint {
```

5.2.2.1. Les données

Pour obtenir des contraintes efficaces, il est nécessaire de rendre les algorithmes incrémentaux (il est impensable de tout recalculer suite à chaque décision prise par l'algorithme de recherche). Nous devons ici stocker les informations suivantes : quels sont les variables qui peuvent ou doivent être affectées à la valeur v qui nous intéresse, et quel est le nombre de variables pouvant ou devant prendre cette valeur.

De plus, pour assurer la consistance des données lors d'un retour arrière de l'algorithme de recherche, nous utiliserons les types *storés* que nous offre Choco :

```
  public StoredBitSet isPossible; // v appartient-elle à x ?
  public StoredBitSet isSure; // x est-elle instanciée à v
  public StoredInt nbPossible;
  public StoredInt nbSure;
```

Enfin, nous stockons dans des variables booléennes sur quelle(s) borne(s) il faut travailler :

```
  public boolean constrainOnInfNumber;
  public boolean constrainOnSupNumber;
```

Le reste des données (et notamment les variables du problème) est directement géré par la super-classe `AbstractLargeIntConstraint`, il suffit donc dans le constructeur de mettre à jour ces variables.

5.2.2.2. Les événements sur la contrainte

La contrainte peut recevoir deux types d'évènements : les évènements liés à la contrainte et les événements liés à une variable en particulier. Nous présentons ici le premier type d'événements.

Les événements liés à la contrainte sont soit un premier réveil `awake` (ce qui donne l'occasion à contrainte d'initialiser certaines données et de faire une première propagation) soit un réveil classique `propagate`.

La méthode awake

Lors du premier réveil, il est possible sans calcul d'affirmer que le nombre d'occurences de la valeur v est compris entre 0 et le nombre de variables (mis à part la variable nbOcc). D'où la définition suivante :

```
  public void awake() throws ContradictionException {
    if (constrainOnInfNumber)
      vars[vars.length - 1].updateSup(vars.length -
1,cIndices[vars.length - 1]);
    if (constrainOnSupNumber)
      vars[vars.length - 1].updateInf(0,cIndices[vars.length - 1]);
    propagate();
  }
```

On notera que cette méthode peut lever une exception `ContradictionException` puisqu'elle modifie des domaines et peut donc entrainer un domaine vide. En effet, `updateInf` (resp. `updateSup`) permet de mettre à jour la valeur minimale (resp. maximale) d'une variable.

Comme le point fixe est atteint (si la contrainte est réveillée pour l'informer de ces modifications, elle ne déduira rien de plus), on précise à ces méthodes l'index de cette contrainte pour éviter qu'elle ne soit appelée suite à ces modifications. Les contraintes stockent ces indices dans le tableau `cIndices` ou sont accessibles via la méthode `getConstraintIdx(int)`. Dans le doute, on peut indiquer la valeur `-1` permettant à la contrainte d'être réveillée suite à une modification dont elle est responsable : dans ce cas, on ne risque que de faire des calculs inutiles.

La méthode propagate

Il reste maintenant à implémenter la méthode `propagate` qui sera lancée lors du réveil de cette contrainte (un tel réveil peut être demandé grâce à la méthode `constAwake(boolean)` des contraintes). Ce réveil a souvent lieu pour mettre à jour les informations de la contrainte (lorsque celle-ci revient dans un état actif par exemple). Nous allons donc dans un premier mettre à jour les données, puis filtrer autant que possible les valeurs impossibles :

```
public void propagate() throws ContradictionException {
  for (int j = 0; j < vars.length - 1; j ++) {
    if (isPossible.get(j)) {
      if (!isSure.get(j) && vars[j].isInstantiatedTo(cste)) {
        isSure.set(j);
        nbSure.add(1);
      } else if (!vars[j].canBeInstantiatedTo(cste)) {
        isPossible.clear(j);
        nbPossible.add(-1);
      }
    }
  }
  filter();
}
```

Cette méthode permet de parcourir l'ensemble des variables (mis à part `nbOcc`) et de mettre à jour les données associées à ces variables (selon qu'elle soit instanciée à v ou qu'elle ne puisse pas être instanciée à v). Puis elle appelle une méthode `filter` qui ne fait pas partie de l'API d'une contrainte mais qui nous permet de traiter à part l'aspect filtrage (ce qui semble être une bonne idée lorsque l'on souhaite par la suite implémenter une version expliquée).

Pour cela, on ajoute les méthodes suivantes :

```
public void checkNbPossible() throws ContradictionException {
   int nbVars = vars.length - 1;
   if (constrainOnInfNumber) {
     vars[nbVars].updateSup(nbPossible.get(),cIndices[nbVars]);
     if (vars[nbVars].getInf() == nbPossible.get()) {
       for (int i = 0 ; i < nbVars ; i ++ ) {
        if (isPossible.get(i))  vars[i].instantiate(cste,cIndices[i]);
       }
     }
   }
}
```

```
public void checkNbSure() throws ContradictionException {
   int nbVars =  vars.length - 1;
   if (constrainOnSupNumber) {
     vars[nbVars].updateInf(nbSure.get(),cIndices[nbVars]);
     if (vars[nbVars].getSup() == nbSure.get()) {
       for (int i = 0 ; i < nbVars ; i ++ ) {
        if (isPossible.get(i) && !vars[i].isInstantiated())
            vars[i].removeVal(cste,cIndices[i]);
       }
     }
   }
}

public void filter() throws ContradictionException {
  checkNbPossible();
  checkNbSure();
}
```

Les méthodes `checkNbPossible` et `checkNbSure` filtre les valeurs en utilisant les principes introduits dans la section sur la sémantique de la contrainte : si le nombre de variables *pouvant* être égal à v est égal au nombre minimal d'occurences que la contrainte doit accepter, alors on peut instancier ces variables.

5.2.2.3. Les événements sur les variables

Le deuxième style d'évenements que peut recevoir une contraint sont les événements liés aux variables (suite à des décisions prises par l'algorithme de recherche ou suite à un filtrage d'une autre contrainte). Les différentes méthodes pour gérer ces événements sont :

- `awakeOnInf` qui réagit sur la modification de la valeur minimale de la variable (elle doit toujours être définie selon la sémantique de la contrainte),
- `awakeOnSup` son symétrique pour la valeur maximale,
- `awakeOnRem` qui réagit sur un retrait d'une valeur entre les deux bornes (à redéfinir),
- `awakeOnInst` qui réagit sur une instantiation d'une de ses variables (à redéfinir),
- `awakeOnRemovals` qui réagit à plusieurs retraits de valeurs : par défaut elle parcourt tous les retraits de valeur de la variable en question et appelle `awakeOnRem`; elle ne nécessite donc pas d'être redéfinie *a priori*,
- enfin `awakeOnVar` qui réagit à une modification de variable sans précision (elle appelle par défaut `propagate`).

Warning:

La sémantique des ces méthodes n'est pas la même si des explications sont utilisées. Dans ce dernier cas, `awakeOnInst` et `awakeOnVar` ne sont pas utilisées, `awakeOnInf`/`awakeOnSup` ne sont utilisées qu'avec des variables de types `BoundIntVar` et `awakeOnRem`/`awakeOnRemovals` ne sont utilisées qu'avec des variables de type `EnumIntVar`. Ceci est dû notamment au stockage des explications pour ces variables.

Nous allons ici redéfinir les quatre première méthodes.

Les méthodes awakeOnInf et awakeOnSup

Ces deux méthodes sont très similaires : si les bornes modifiées ne permettent plus l'instantiation de la variable à la valeur v, il faut mettre à jour les données. De plus, si ces

bornes deviennent égales à cette valeur v, il faut vérifier qu'il est possible d'affecter une variable de plus à v (notamment pour les variables de type `BoundIntVar` qui ne permettent pas de retirer les valeurs au milieu de leurs bornes), et dans le cas contraire la borne est encore décalée pour retirer cette valeur du domaine. Par exemple, dans le cas de la borne inférieure, cela donne :

```
public void awakeOnInf(int idx) throws ContradictionException {
  int nbVars = vars.length - 1;
  if (idx < nbVars) {
    if (isPossible.get(idx)) {
      if (vars[idx].getInf() > cste) {
        isPossible.clear(idx);
        nbPossible.add(-1);
        checkNbPossible();
      } else if (vars[idx].getInf() == cste &&
             !(isSure.get(idx)) &&
             constrainOnSupNumber &&
             nbSure.get() == vars[nbVars].getSup()) {
        vars[idx].updateInf(cste + 1, cIndices[idx]);
      }
    }
  } else
    checkNbPossible();
}
```

De plus, dans le cas où c'est nbOcc qui est modifiée, on appelle `checkNbPossible` pour vérifier qu'il y a toujours assez de variables pouvant être affectées à v (dans le cas de la borne supérieure, la méthode `checkNbSure` aurait été appelée).

La méthode awakeOnRem

Cette méthode est plus simple : s'il s'agit d'un retrait de la valeur v, il faut mettre à jour les données, et vérifier qu'il y a toujours suffisamment de variables pouvant être instanciées à v.

```
public void awakeOnRem(int idx, int x) throws ContradictionException {
  int nbVars = vars.length - 1;
  if (idx < nbVars && x == cste && isPossible.get(idx)
    && vars[idx].hasEnumeratedDomain()) {
    isPossible.clear(idx);
    nbPossible.add(-1);
    checkNbPossible();
  }
}
```

La condition `vars[idx].hasEnumeratedDomain()` permet tout simplement de vérifier qu'il ne s'agit pas d'une variable de type `BoundIntVar` car dans ce cas, il ne faut pas prendre en compte cet événement puisqu'elle ne peut pas stocker les informations à propos des retraits de valeurs entre les deux bornes. De plus, s'il s'agit de la variable nbOcc, il n'y a rien à faire car nous ne nous intéressons uniquement aux bornes de cette variable.

La méthode awakeOnInst

Cette dernière méthode est un mélange de `awakeOnInf` et `awakeOnSup` : si la valeur v était possible mais pas sur, il faut vérifier si la variable est instanciée à v ou non pour mettre à jour correctement les données. S'il s'agit de la variable nbOcc (et non de xi), il faut vérifier qu'il y a suffisamment de variables pouvant être affectée à v et pas trop déjà instantiées à v :

```
  public void awakeOnInst(int idx) throws ContradictionException {
    int nbVars = vars.length - 1;
    if (idx < nbVars && isPossible.get(idx) && !(isSure.get(idx))) {
      if (vars[idx].getValue() == cste) {
        isSure.set(idx);
        nbSure.add(1);
        checkNbSure();
      } else {
        isPossible.clear(idx);
        nbPossible.add(-1);
        checkNbPossible();
      }
    } else {
      checkNbPossible();
      checkNbSure();
    }
  }
```

5.2.2.4. La construction de la contrainte

Pour créer la contrainte, il faut bien entendu fournir un constructeur permettant d'initialiser variables, données, etc :

```
  public Occurence(IntVar[] lvars, int occval, boolean onInf, boolean
onSup) {
    super(lvars.length);
    this.cste = occval;  // L valeur étudiée v
    this.constrainOnInfNumber = onInf;  // Propagation sur le nombre
minimal
    this.constrainOnSupNumber = onSup;  // Propagation sur le nombre
maximal

    this.problem = lvars[0].getProblem();  // Stocke le problème associé

    int nbVars = lvars.length;
    // Détermine l'environnement utilisés pour  les données storées
    Environment envi = lvars[0].getProblem().getEnvironment();
    isPossible = new StoredBitSet(envi, nbVars - 1);
    isSure = new StoredBitSet(envi, nbVars - 1);
    nbPossible = new StoredInt(envi, 0);
    nbSure = new StoredInt(envi, 0);

    // Affectation des variables et mis à jour des données
    for (int i = 0; i < (nbVars - 1); i++) {
      vars[i] = (IntDomainVar) lvars[i];
      if (vars[i].canBeInstantiatedTo(occval)) {
        isPossible.set(i);
        nbPossible.add(1);
      }
      if (vars[i].isInstantiatedTo(occval)) {
        isSure.set(i);
```

```
        nbSure.add(+1);
      }
    }
    vars[nbVars - 1] = (IntDomainVar) lvars[nbVars - 1];
  }
```

5.2.2.5. Quelques méthodes supplémentaires

Quelques méthodes peuvent être encore nécessaire, comme `isSatisfied` qui permet de vérifier si la contrainte est vérifiée (une fois toutes les variables instantiées). Il suffit ici de compter le nombre d'apparitions de la valeur v sur l'ensemble des variables :

```
  public boolean isSatisfied() {
    int nbVars = vars.length - 1;
    int cptVal = 0;
    for (int i = 0; i < nbVars; i++) {
      if(vars[i].getValue() == cste) cptVal++;
    }
    if (constrainOnInfNumber && constrainOnSupNumber)
      return cptVal == vars[nbVars].getValue();
    else if (constrainOnInfNumber)
      return cptVal >= vars[nbVars].getValue();
    else return cptVal <= vars[nbVars].getValue();
  }
```

5.2.3. Implémentation d'une contrainte avec explications

L'implémentation d'une contrainte avec explications demandent plus d'information. Nous ne détaillerons donc pas l'implémentation complète de la contrainte. Dans la version classique (sans explications), le filtrage a bien été séparé du reste dans les méthodes `checkNbPossible` et `checkNbSure`; nous devrons donc principalement modifier ces méthodes de sorte à y générer les explications justifiant les décisions prises. De plus, les contraintes expliquées sont utilisées dans un cadre dynamique : des contraintes sont ajoutées et d'autres retirées, ce qui nécessite de maintenir les données. Il existe donc maintenant deux types de méthodes réagissant sur les événements liés aux variables : les méthodes modifiant les données `updateDataStructuresOnX` et les méthodes lançant le filtrage en lui-même. Nous verrons comment implémenter certaines d'entre elles. Enfin, des méthodes utiles peuvent (et devraient) être implémentées comme `whyIsTrue` ou `whyIsFalse`.

Warning:
Nous supposons ici que l'on souhaite hériter de la contrainte sans explication. Ceci n'est pas obligatoire, mais peut être utile lorsque l'on ne souhaite pas dupliquer la structure de donnée (qui peut être complexe dans certains cas !).

5.2.3.1. Expliquer le filtrage

Pour utiliser les explications, il faut pouvoir déterminer pour chaque valeurs filtrées ce qui permet de justifier ce retrait. Ainsi, si l'on sait quels propriétés (borne inférieure, supérieure, valeur absente d'un domaine...) sont à l'origine du retrait, il est possible de générer une explication en faisant l'union des explications de ces propriétés (toute

modification d'un domaine d'une variable stocke une explication permettant de justifier celle-ci).

Prenons le cas de `checkNbPossible`. La première chose que fait cette méthode, c'est de mettre à jour la borne supérieure de nbOcc pour que celle-ci soit inférieure ou égale au nombre de variables pouvant être instanciées à v. Pour justifier cette valeur, il suffit de justifier le nombre de variables ne pouvant pas prendre cette valeur (le nombre de variables est fixe), c'est-à-dire de justifier que ces variables ne peuvent plus être instantiée à cette valeur v. De plus, l'explication doit contenir la contrainte en cours d'exécution qui est responsable de ce filtrage (grâce à la ligne suivant l'initialisation de l'explication dans le code). La méthode commencera donc comme suit :

```
  public void checkNbPossible() {
    if (this.constrainOnInfNumber) {
      Explanation expl = new GenericExplanation(problem);
      ((PalmConstraintPlugin) this.hook).self_explain(expl);

      for (int i = 0; i < this.vars.length - 1; i++) {
        if (!this.isPossible.get(i)) {
          ((PalmIntVar) this.vars[i]).
              self_explain(PalmIntDomain.VAL, this.cste, expl);
        }
      }

      ((PalmIntVar) this.vars[this.vars.length - 1]).updateSup(
          this.nbPossible.get(), cIndices[this.vars.length - 1],
expl.copy());
```

Ensuite, si ce nombre de variables pouvant être égal à v est égal au nombre minimal d'occurences que doit accepter la contrainte, alors il faut instantier toutes ces variables. L'explication sera alors la même si ce n'est qu'on lui ajoutera la justification de cette borne inférieure (celle de la variable nbOcc) :

```
      if (this.vars[this.vars.length - 1].getInf() ==
this.nbPossible.get()) {
        ((PalmIntVar) this.vars[this.vars.length - 1]).
            self_explain(PalmIntDomain.INF, expl);
        for (int i = 0; i < this.vars.length - 1; i++) {
          if (this.isPossible.get(i)) {
            ((PalmIntVar) this.vars[i]).instantiate(this.cste,
                cIndices[i], expl.copy());
          }
        }
      }

    }
    this.checkPossible = false;
  }
```

Nous expliquerons plus loin à quoi correspond la dernière ligne de cette méthode.

5.2.3.2. Maintien des structures

Dans le cas de retour arrière simple, il n'est pas gênant que les événements sur les variables ne soient pas traitées de manière synchrone : si une contradiction est levée, tous

les événements sont nettoyés. Dans un cadre dynamique, ceci n'est plus possible : il est donc important de garantir que les structures de données sont toujours à jour. Pour cela, des méthodes sont appelés de manière synchrone dès qu'un domaine est modifié pour donner une chance aux contraintes de maintenir leurs structures de données.

Dans le cas présent, maintenir la structure de données revient à mettre à jour les variables `nbPossible`, `isPossible`... Si l'on considère le cas d'une modification de borne inférieure par exemple, si la borne devient supérieure à la valeur v il faut mettre à jour le nombre de variables possibles. De même si la variable est instanciée à la valeur v il faut mettre à jour le nombre de variables sures.

```
public void updateDataStructuresOnConstraint(int idx, int select,
    int newValue, int oldValue) {
  switch (select) {
    case PalmIntDomain.INF:
      if (idx < this.getNbVars() - 1) {
        if ((newValue > this.cste) && (this.isPossible.get(idx))) {
          this.isPossible.clear(idx);
          this.nbPossible.add(-1);
          this.checkPossible = true;
          if (this.isSure.get(idx)) {
            this.isSure.clear(idx);
            this.nbSure.add(-1);
            this.checkSure = true;
          }
        }

        if (this.vars[idx].isInstantiated()) {
          int val = this.vars[idx].getValue();
          if ((val == this.cste) && (!this.isSure.get(idx))) {
            this.isSure.set(idx);
            this.nbSure.add(1);
            this.checkSure = true;
          }
        }
      } else {
        this.checkSure = true;
        this.checkPossible = true;
      }
      break;
```

On notera deux points justifiant ce code :

- si la valeur n'est possible alors qu'elle était sûre, unne cotradiction sera levée; cependant il est important de maintenir la structure à jour, car rien ne garantit que l'on remettra cette valeur (il ne s'agit plus d'un retour arrire classique !),
- si la variable est instanciée à une autre valeur que v, alors soit cette valeur est plus grande que v et donc le nombre des variables possibles a été mis à jour, soit cette valeur est plus petite que v, et cela signifie qu'un événement `awakeOnSup` sera lancé aussi permettant de mettre à jour cette variable (il ne faut donc pas mettre à jour cette variable plusieurs fois !).

Les deux dernières lignes, entre autres, mettent en avant deux nouvelles variables. Ces variables permettent de gérer aussi rapidement que possible le filtrage en évitant de

refaire des tests déjà effectués (si l'on sait que rien ne sera filtré, il est inutile d'essayer).

La deuxième étape consiste à en faire de même pour la restauationde valeurs. Dans ce cas, il faut incrémenter le nombre de variables possibles ou sures si nécessaire. De plus, si une variable n'est plus instantiée (on lui ajoute une deuxième valeur) il faut modifier le nombre de variables sures. Ceci donne donc :

```
  public void updateDataStructuresOnRestoreConstraint(int idx, int
select,
      int newValue, int oldValue) {
    switch (select) {
      case PalmIntDomain.INF:
        if (idx < this.getNbVars() - 1) {
          if ((newValue <= this.cste) && (oldValue > this.cste)
              && (this.vars[idx].getSup() >= newValue)) {
            this.isPossible.set(idx);
            this.nbPossible.add(1);
            this.checkPossible = true;
            if (this.vars[idx].isInstantiated()) {
              int val = this.vars[idx].getValue();
              if ((val == this.cste) && (!this.isSure.get(idx))) {
                this.isSure.set(idx);
                this.nbSure.add(1);
                this.checkSure = true;
              }
            }
          }
          if ((this.isSure.get(idx))
              && (this.vars[idx].getInf() != this.cste)) {
            this.isSure.clear(idx);
            this.nbSure.add(-1);
            this.checkSure = true;
          }
        } else {
          this.checkPossible = true;
          this.checkSure = true;
        }
        break;
```

5.2.3.3. Exécution du filtrage

Le maintien des structures ayant spécifié les filtrages à lancer, le travail est mainteant beaucoup plus simple lors des réveils liés ax variables. Cependant, il reste des cas à traiter. Si l'on reprend le cas de la borne inférieure, si la borne devient égale à v, il est nécessaire de vérifier qu'il est possible que cette variable prenne cette valeur (c'est-à-dire qu'il n'y a pas déjà trop de variables instanciées à cette valeur) : il s'agit d'un filtrage de provenant pas directement de la structure de donnée, c'est pour cela qu'il est nécessaire de le traiter à part. On obtient alors le code suivant :

```
  public void awakeOnInf(int idx) {
    if (idx < this.getNbVars()) {
      if (this.checkPossible)
        this.checkNbPossible();
      if (this.checkSure)
        this.checkNbSure();
      if ((this.isPossible.get(idx)) &&
```

```
          (this.vars[idx].getInf() == this.cste))
        if ((!this.isSure.get(idx)) && (this.constrainOnSupNumber) &&
            (this.nbSure.get() == this.vars[this.vars.length -
1].getSup())) {
          Explanation expl = new GenericExplanation(problem);
          ((PalmConstraintPlugin) this.hook).self_explain(expl);
          ((PalmIntVar) this.vars[idx]).
              self_explain(PalmIntDomain.INF, expl);
          ((PalmIntVar) this.vars[this.vars.length - 1]).
              self_explain(PalmIntDomain.SUP, expl);
          for (int i = 0; i < this.getNbVars() - 1; i++) {
            if (this.isSure.get(i))
              ((PalmIntVar) this.vars[i]).
                  self_explain(PalmIntDomain.DOM, expl);
          }
          ((PalmIntVar) this.vars[idx]).
              updateInf(this.cste + 1, cIndices[idx], expl);
        }
    } else {
      this.checkNbPossible();
    }
  }
```

Le même style de code devra être développé pour `awakeOnSup`, `awakeOnRem` ou `awakeOnRemovals`.

5.2.3.4. Les réveils lors de la restauration

Lorsqu'une contrainte est retirée, certaines valeurs peuvent être restaurées dans les domaines des variables. Il faut donc vérifier si ces valeurs sont consistantes avec la contrainte. Ici, il est difficile de savoir quelle partie du filtrage doit être vérifiée, les fonctions appelle donc la méthode `filter()` qui elle-même vérifie à la fois les propriétés sur le nombres de variables possibles et le nombre de nécessaires :

```
  public void awakeOnRestoreInf(int idx) throws ContradictionException {
    this.filter();
  }

  public void awakeOnRestoreSup(int idx) throws ContradictionException {
    this.filter();
  }

  public void awakeOnRestoreVal(int idx, int val) throws
ContradictionException {
    this.filter();
  }

  public void awakeOnRestoreVal(int idx, IntIterator it)
      throws ContradictionException {
    for (; it.hasNext();) {
      awakeOnRestoreVal(idx, it.next());
    }
  }
```

On notera que puisque l'on hérite de la version non expliquée, il est nécessaire de définir une méthode `awakeOnRestoreVal(int, IntIterator)` puisqu'aucune superclasse ne la définit.

5.2.3.5. Quelques autres méthodes

Outre la déclaration des données supplémentaires, la construction de la contrainte version expliquée ne demande pas d'information particulière si ce n'est l'initialisation d'un *plugin* permettant de stocker les informations relatifs aux explications :

```
  private boolean checkPossible = false, checkSure = false;

  public PalmOccurence(IntVar[] lvars, int occval, boolean onInf,
boolean onSup) {
    super(lvars, occval, onInf, onSup);
    this.hook = new PalmConstraintPlugin(this);
  }
```

Outre les méthodes ues jusqu'ici (obligatoires pour le bon fonctionnement de la contrainte), on peut définir les fonctions `whyIsTrue` et `whyIsFalse` permettant de savoir pourquoi une contrainte est forcément fausse, ou pourquoi elle est forcément vérifiée.

Enfin, comme il s'agit d'une contrainte expliquée, la classe implémente les interfaces suivantes : `PalmConstraint` et `PalmIntVarListener`.

FIXME (Guillaume):

Voir si on parle de `takeIntoAccountStatusChange(int)` !?

5.3. Tutoriel : Expliquer une contrainte

L'objectif de ce tutoriel est de présenter comment expliquer une contrainte c'est-à-dire comment justifier chacune des décisions de filtrage prises par une contrainte. Ceci permet par la suite d'utiliser la contrainte avec des algorithmes dédiés mais aussi d'avoir des explications sur le comportement du solveur (absence de solutions notamment).

Pour cela, nous montrerons comment implémenter une contrainte de supériorité permettant d'assurer qu'une variable x est inférieur à une variable y. Puis nous monterons comment cela se traduit dans le cas de contraintes plus complexes comme les combinaisons linéaires.

5.3.1. Construction d'une contrainte expliquée

Choco propose plusieurs classes abstraites dont les contraintes que l'on crée peuvent hériter pour éviter de ne devoir coder des méthodes *non métier*. Dans notre cas, on souhaite modéliser la contrainte x#y+c. Il s'agit donc d'une classe binaire (deux variables) qui héritera de la classe `AbstractPalmBinIntConstraint`. Pour construire la contrainte il ne reste plus qu'à initialiser ces données (les variables x et y, et la valeur c) et ajouter la ligne `this.hook = new PalmConstraintPlugin(this);` pour créer une entité dont le but est de stocker les informations relatives aux explications (les valeurs retirées d'un domaine à cause de cette contrainte par exemple).

```
public class PalmGreaterOrEqualXYC extends AbstractPalmBinIntConstraint
{
  protected final int cste;

  public PalmGreaterOrEqualXYC(IntVar v0, IntVar v1, int cste) {
    this.v0 = v0;
    this.v1 = v1;
    this.cste = cste;
    this.hook = new PalmConstraintPlugin(this);
  }

  public String toString() {
    return this.v0 + " >= " + this.v1 + " + " + this.cste;
  }
```

On en profite pour définir une méthode `toString` permettant d'afficher la contrainte de manière lisible.

5.3.2. Propagation sur la contrainte

Tout d'abord, nos n'avons pas besoin de comportement particulier pour la première propagation de la contrainte, nous ne définissons donc ici que la méthode `propagate()`.

De plus, le premier filtrage revient à une situation où les bornes sont modifiées (on peut considérer que l'on passe de [-#,#] à un intervalle plus petit). La méthode se réduit donc au code suivant :

```
  public void propagate() {
    this.awakeOnInf(1);
    this.awakeOnSup(0);
  }
```

C'est tout ce que nous avons à faire ici.

5.3.3. Propagation sur les variables

A priori, `awakeOnInf` et `awakeOnSup` réagissent sur des modifications de domaine de variables non énumérées tandis que `awakeOnRem` résgit sur des variables énumérées. Cependant, cela n'implique en rien que l'autre variable a le même type ! Il faut donc bien prévoir ces deux types de variables.

En effet, nous pourrions gérer les variables énumérées comme les variables sur les bornes. Cependant, s'il s'agit de variables énumérées, il est souhaitable d'avoir les explications les plus précises possibles étant donné que le nombre de valeurs doit être relativement faible. Les méthodes de filtrage auront donc deux comportements selon que les variables soient énumérées ou non :

- si la variable dont on filtre des valeurs est énumérée, les explications pour chaque retrait doivent être aussi précises que possible puisque de toute manière chaque valeur retirée doit être expliquée séparément,
- par contre, si la variable dont on filtre des valeurs n'est pas énumérée, il suffit d'expliquer la nouvelle borne (dans le cas contraire, on risque de stocker un nombre

d'explications trop important).

Par exemple, considérons le réveil sur la modification de la borne inférieure d'une variable. On doit réagir s'il s'agit de la variable y si elle yinf+c est supérieur strictement à xinf (on veut x#y+c). Cela donne alors :

```
  public void awakeOnInf(int idx) {
    if ((idx == 1) && (v1.getInf() + this.cste > v0.getInf())) {
      if (v0.hasEnumeratedDomain()) {
        int[] values = ((PalmIntVar) v0).getAllValues();
        choco.palm.explain.Explanation expl =
            new
choco.palm.explain.GenericExplanation(this.getProblem());
        ((PalmConstraintPlugin) this.getPlugIn()).self_explain(expl);
        int index = 0;
        int min = ((PalmIntDomain) v1.getDomain()).getOriginalInf();
        while (index < values.length &&
            values[index] < v1.getInf() + this.cste) {
          for (int i = min; i <= values[index] - this.cste; i++)
            ((PalmIntVar) this.v1).self_explain(PalmIntDomain.VAL, i,
expl);
          ((PalmIntVar) this.v0).removeVal(values[index],
              this.cIdx0, expl.copy());
          min = values[index] + 1 - this.cste;
          index++;
        }
      } else {
        choco.palm.explain.Explanation expl =
            new
choco.palm.explain.GenericExplanation(this.getProblem());
        ((PalmConstraintPlugin) this.getPlugIn()).self_explain(expl);
        ((PalmIntVar) this.v1).self_explain(PalmIntDomain.INF, expl);
        ((PalmIntVar) this.v0).updateInf(this.v1.getInf() + this.cste,
            this.cIdx0, expl);
      }
    }
  }
```

Étudions plus précisément ce code selon le type de la variable x :

- si x est énumérée, on veut une explication par valeur retirée, donc il est intéressant de fournir une explication la plus précise possible; dans ce cas, après avoir ajouté dans l'explication la contrainte actuelle (en utilisant la méthode `getPlugIn()`), on calcule pour chaque valeur v à retirer du domaine de x une explication; plus la valeur sera importante, plus l'explication contiendra de justification de retraits de valeurs de y;
- par contre si x n'est pas énumérée (on ne stocke que les bornes), il ne faut pas calculer toutes les explications sinon on risque d'utiliser beaucoup trop de mémoire pour stocker toutes les explications; on peut donc directement utiliser l'explication de la borne inférieure de y.

On notera que l'explication de la plus grande des valeurs retirées de x sera la même dans les deux cas : ce n'est que pour les valeurs intermédiaires que le résultat est différent.

La méthode `awakeOnSup` sera définie de manière similaire : il faut juste parcourir les valeurs dans l'ordre décroissant dans le cas d'une variable énumérée.

Pour ce qui est de awakeOnRem, il ne s'agit que d'un cas particulier des méthodes précédentes. Si l'on retire une valeur de y, il faut vérifier la bornes inférieure de x; si on reetir une valeur de x, il faut vérifier la borne supérieure de y. Cela donne :

```
public void awakeOnRem(int idx, int v) {
  if (idx == 0) {
    this.awakeOnSup(0);
  } else {
    this.awakeOnInf(1);
  }
}
```

5.3.4. Propagation de la restauration

Lorsque les explications sont utilisées (notamment dans un cadre dynamique ou l'algorithme mac-dbt), des valeurs sont restaurées dans les domaines des variables lorsqu'une contrainte est retirée. Il faut alors vérifier que cette valeur est consitante ou non.

Warning:

Rien ne garantit lors de l'appel de la fonction que la valeur ou borne restaurée n'a pas déjà été supprimée. Le but est juste de vérifier si l'on peut retirer la valeur, et non pas de filtrer à cause de cette apparition de la valeur !

Dans le cas de l'inégalité, cela revient à vérifier la borne concernée. Par exemple, si la borne inférieure de x a été restaurée, il faut vérifier qu'elle est consistante. Pour cela il suffit de considérer que la borne inférieure de y a changé. Ainsi, on obtient les méthodes suivantes :

```
public void awakeOnRestoreInf(int idx) {
  if (idx == 0) this.awakeOnInf(1);
}

public void awakeOnRestoreSup(int idx) {
  if (idx == 1) this.awakeOnSup(0);
}
```

Le même raisonnement peut être tenu pour la restauration de valeur et non de bornes :

```
public void awakeOnRestoreVal(int idx, int val) {
  if (idx == 1) {
    this.awakeOnSup(0);
  } else {
    this.awakeOnInf(1);
  }
}
```

Cette étape de propagation après restauration est importante car elle garantit de ne pas trouver trop de solutions : si trop de solutions sont trouvées, il est fort probable qu'une phase de propagation lors de la restauration est mal codée.

5.3.5. Quelques méthodes utiles

whyIsTrue, isEntailed...