MCSP3
Easy Modeling for Everybody

Release Candidate

Christophe Lecoutre
CRIL CNRS, UMR 8188
University of Artois, France
lecoutre@cril.fr

February 23, 2017

1 Introduction

We now propose a complete chain of production for solving combinatorial constrained problems. The two main ingredients are:

- MCSP3: a Java-based API for modeling constrained problems in a natural and declarative way
- XCSP3: an intermediate format used to represent problem instances while preserving structure

A shown on the left of Figure 1, the user has to:

- write a model using the Java modeling API called MCSP3
- furnish the data (in JSON format) for some specific instances
- compile model and data in order to get XCSP3 files
- solve the XCSP3 instance(s) using solvers like for example Choco or OscaR

The complete chain of production, called MCSP3-XCSP3, has many advantages:

- JSON, Java and XML are robust mainstream technologies
- Using JSON for data permits to have a unified notation, easy to read for both humans and machines
- using Java 8 for modeling permits the user to avoid learning again a new programming language
- Using a coarse-grained XML structure permits to have quite readable problem descriptions, easy to read for both humans and machines

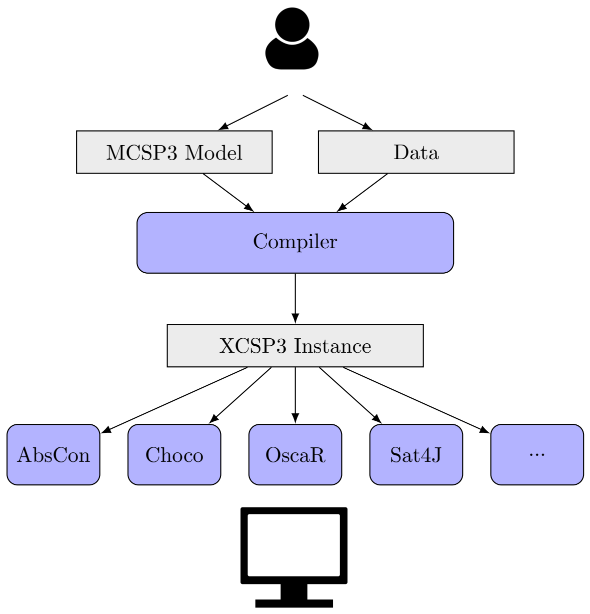


Figure 1: Complete chain of production for modeling and solving combinatorial constrained problems, based on MCSP3 and XCSP3.

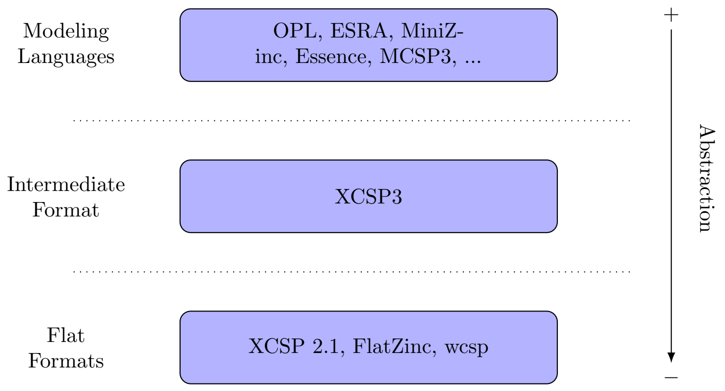


Figure 2: Modeling Languages and Formats

Remark. Using JCSP3 for representing instances, i.e., JSON instead of XML, is possible but has some (minor) drawbacks, as explained in an appendix of XCSP3 Specifications.

Currently, this is a release candidate of the API MCSP3 (with its compiler). It is available on github: https://github.com/xcsp3team/XCSP3-Java-Tools, in package `modeler`.

2 Single Problems

We propose to start discovering the API with some case studies.

2.1 A Simple Riddle

Remember that when you were young, you were used to play at riddles, some of them having a mathematical background as for example:

> Which sequence of successive four integer numbers sum up to 14?

If you were already familiar with Mathematics, maybe you were able to formalize this ridlle by:

- introducing four integer variables:
  - $x_1 \in \mathbb{N}$, $x_2 \in \mathbb{N}$, $x_3 \in \mathbb{N}$, $x_4 \in \mathbb{N}$
- introducing the following mathematical relations (constraints):
  - $x_1 + 1 = x_2$
  - $x_2 + 1 = x_3$
  - $x_3 + 1 = x_4$
  - $x_1 + x_2 + x_3 + x_4 = 14$

This is a CSP (Constraint Satisfaction problem) instance, involving four integer variables, three binary constraints and one quaternary constraint.

After a rough analysis, we can decide to set 0 as lower bound and 14 as upper bound for the possible values of the integer variables. We then obtain the following model in MCSP3:

```
class Riddle implements ProblemAPI {

  public void model() {
    Var x1 = var("x1", dom(range(15)));
    Var x2 = var("x2", dom(range(15)));
    Var x3 = var("x3", dom(range(15)));
    Var x4 = var("x4", dom(range(15)));
    equal(add(x1, 1), x2);
    equal(add(x2, 1), x3);
    equal(add(x3, 1), x4);
    equal(add(x1, x2, x3, x4), 14);
  }
}
```

As you can observe, you just need to build a Java class that implements the interface `ProblemAPI` and that contains a method `model()`. Within this method, to declare a stand-alone variable, you have to call one of the following methods `var()`:

```
Var var(String id, XDomInteger dom, String note)
Var var(String id, XDomInteger dom)

VarSymbolic var(String id, XDomSymbolic dom, String note)
VarSymbolic var(String id, XDomSymbolic dom)
```

As you can easily imagine, `Var` and `VarSymbolic` are the classes in the API used to represent respectively integer and symbolic variables[1]. The specified id is the identification of the variable; it is mandatory and must be unique. The optional parameter `note` is a short comment that can be useful to describe the role of the variable. For example, if we want "x1 is extraordinary" as comment in the XCSP3 file that can be generated from this MCSP3 model, we have to replace the first statement in Method `model()` above with:

```
Var x1 = var("x1", dom(range(15)),"x1 is extraordinary");
```

Now, the question is: how can we build domains (objects from classes `XDomInteger` and `XDomSymbolic`)? The answer is that you just have to call one of the following methods `dom()`:

```
XDomInteger dom(int value1, int... otherValues)
XDomInteger dom(int[] values)
XDomInteger dom(Collection<Integer> values)
XDomInteger dom(int[][] m)
XDomInteger dom(Range range)

XDomSymbolic dom(String value1, String... otherValues)
XDomSymbolic dom(String[] values)
```

where the fifth method requires an object Range that can be obtained with one of the following methods `range()`:

```
Range range(int minIncluded, int maxIncluded, int step)
Range range(int minIncluded, int maxIncluded)
Range range(int length)
```

As an illustration, the following table shows which domains are obtained for various calls. It is important to note that integer domains are systematically sorted (without any two occurences of the same value).

[1]Support for real, set and graph variables will be proposed in a future version.

Call	Domain
`dom(0,1)`	$\{0,1\}$
`dom(1,2,5,10)`	$\{1,2,5,10\}$
`dom(1,2,5,10,2,4)`	$\{1,2,4,5,10\}$
`dom(new int[] {2,3,4})`	$\{2,3,4\}$
`dom(new int[][] {{1,3},{2,2},{4,6}})`	$\{1,2,3,4,6\}$
`dom(range(5))`	$\{0,1,2,3,4\}$
`dom(range(10,15))`	$\{10,11,12,13,14,15\}$
`dom(range(1,10,3))`	$\{1,4,7,10\}$
`dom("red","green","blue")`	{“red”,”green”,”blue”}

Now, let us consider the constraints. Here, we only use intensional constraints. To build a constraint `intension`, you have to call the method `intension()` with an object `XNodeParent` as parameter for representing the tree-shaped Boolean expression. As shown below, all classical operators can be used:

```
CtrEntity intension(XNodeParent<IVar> tree)

XNodeParent<IVar> neg(Object operand)
XNodeParent<IVar> abs(Object operand)
XNodeParent<IVar> add(Object... operands)
XNodeParent<IVar> sub(Object operand1, Object operand2)
XNodeParent<IVar> mul(Object... operands)
XNodeParent<IVar> div(Object operand1, Object operand2)
XNodeParent<IVar> mod(Object operand1, Object operand2)
XNodeParent<IVar> sqr(Object operand) {
XNodeParent<IVar> pow(Object operand1, Object operand2)
XNodeParent<IVar> min(Object... operands)
XNodeParent<IVar> max(Object... operands)
XNodeParent<IVar> dist(Object operand1, Object operand2)

XNodeParent<IVar> lt(Object operand1, Object operand2)
XNodeParent<IVar> le(Object operand1, Object operand2)
XNodeParent<IVar> ge(Object operand1, Object operand2)
XNodeParent<IVar> gt(Object operand1, Object operand2)
XNodeParent<IVar> ne(Object... operands)
XNodeParent<IVar> eq(Object... operands)

XNode<IVar> set(Object... operands)
XNode<IVar> set(int[] operands)
XNodeParent<IVar> in(Object var, Object set)

XNodeParent<IVar> not(Object operand)
XNodeParent<IVar> and(Object... operands)
XNodeParent<IVar> or(Object... operands)
XNodeParent<IVar> xor(Object... operands)
XNodeParent<IVar> iff(Object... operands)
XNodeParent<IVar> imp(Object operand1, Object operand2)
```

```
XNodeParent<IVar> ifThenElse(Object operand1, Object operand2, Object operand3)
```

The precise semantics of these operators is given by Table 1.

At this point, one may wonder if the constraints of our model should have been declared as:

```
intension(eq(add(x1, 1), x2));
intension(eq(add(x2, 1), x3));
intension(eq(add(x3, 1), x4));
intension(eq(add(x1, x2, x3, x4), 14));
```

This is correct, indeed. However, there are some methods in the API that allow us to simplify the expression of some intensional constraints. This is shown by the following table.

Normal Form	Simplified Form
`intension(lt(...))`	`lessThan(...)`
`intension(le(...))`	`lessEqual(...)`
`intension(ge(...))`	`greaterEqual(...)`
`intension(gt(...))`	`greaterThan(...)`
`intension(eq(...))`	`equal(...)`
`intension(ne(...))`	`notEqual(...)`
`intension(imp(...))`	`imply(...)`
`intension(in(...))`	`belong(...)`

Once you have an MCSP3 model, you can compile it in order to get an XCSP3 file that can be given to a constraint solver. The command is as follows[2]:

```
java org.xcsp.modeler.Compiler Riddle
```

The content of the XCSP3 file is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <var id="x1"> 0..14 </var>
    <var id="x2"> 0..14 </var>
    <var id="x3"> 0..14 </var>
    <var id="x4"> 0..14 </var>
  </variables>
  <constraints>
    <intension> eq(add(x1,1),x2) </intension>
    <intension> eq(add(x2,1),x3) </intension>
    <intension> eq(add(x3,1),x4) </intension>
    <intension> eq(add(x1,x2,x3,x4),14) </intension>
  </constraints>
  </instance>
```

[2]You may need to prefix the name of the class with the relevant package information. For example, if Riddle is in package `problems.puzz`, you have to write `package.puzz.Riddle`

Operation	Arity	Syntax	Semantics
Arithmetic (integer operands and integer result)			
Opposite	1	neg(x)	$-x$
Absolute Value	1	abs(x)	$\|x\|$
Addition	$r \geq 2$	add$(x_1, \ldots, x_r)$	$x_1 + \ldots + x_r$
Subtraction	2	sub(x, y)	$x - y$
Multiplication	$r \geq 2$	mul$(x_1, \ldots, x_r)$	$x_1 * \ldots * x_r$
Integer Division	2	div(x, y)	x/y
Remainder	2	mod(x, y)	$x\%y$
Square	1	sqr(x)	x^2
Power	2	pow(x, y)	x^y
Minimum	$r \geq 2$	min$(x_1, \ldots, x_r)$	$\min\{x_1, \ldots, x_r\}$
Maximum	$r \geq 2$	max$(x_1, \ldots, x_r)$	$\max\{x_1, \ldots, x_r\}$
Distance	2	dist(x, y)	$\|x - y\|$
Relational (integer operands and Boolean result)			
Less than	2	lt(x,y)	$x < y$
Less than or equal	2	le(x, y)	$x \leq y$
Greater than or equal	2	ge(x, y)	$x \geq y$
Greater than	2	gt(x, y)	$x > y$
Different from	2	ne(x, y)	$x \neq y$
Equal to	$r \geq 2$	eq$(x_1, \ldots, x_r)$	$x_1 = \ldots = x_r$
Set (a_i: integers, s: set of integers (no variable permitted), x: integer operand)			
Empty set	0	set()	$\emptyset$
Non-empty set	$r > 0$	set$(a_1, \ldots, a_r)$	$\{a_1, \ldots, a_r\}$
Membership	2	in(x, s)	$x \in s$
Logic (Boolean operands and Boolean result)			
Logical not	1	not(x)	$\neg x$
Logical and	$r \geq 2$	and$(x_1, \ldots, x_r)$	$x_1 \wedge \ldots \wedge x_r$
Logical or	$r \geq 2$	or$(x_1, \ldots, x_r)$	$x_1 \vee \ldots \vee x_r$
Logical xor	$r \geq 2$	xor$(x_1, \ldots, x_r)$	$x_1 \oplus \ldots \oplus x_r$
Logical equivalence	$r \geq 2$	iff$(x_1, \ldots, x_r)$	$x_1 \Leftrightarrow \ldots \Leftrightarrow x_r$
Logical implication	2	imp(x, y)	$x \Rightarrow y$
Control			
Alternative	3	if(b, x, y)	value of x, if b is true, value of y, otherwise

Table 1: Operators on integers that can be used to build predicates. Boolean values *false* and *true* are represented by integer values 0 and 1.

The variables of our problem have been declared independently, but it is possible to declare them in a one-dimensional array. This gives:

```
class Riddle2 implements ProblemAPI {

  public void model() {
    Var x[] = array("x", size(4), dom(range(15)));

    equal(add(x[0], 1), x[1]);
    equal(add(x[1], 1), x[2]);
    equal(add(x[2], 1), x[3]);
    equal(add(x[0], x[1], x[2], x[3]), 14);
  }
}
```

and the XCSP3 file obtained after compilation is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" size="[4]"> 0..14 </array>
  </variables>
  <constraints>
    <intension> eq(add(x[0],1),x[1]) </intension>
    <intension> eq(add(x[1],1),x[2]) </intension>
    <intension> eq(add(x[2],1),x[3]) </intension>
    <intension> eq(add(x[0],x[1],x[2],x[3]),14) </intension>
  </constraints>
</instance>
```

Here, we declare a one-dimensional array of variables: its id is "x", its size (length) is 4 and each of its variables has $\{0, 1, \ldots, 14\}$ as domain. Note that indexing starts at 0. The methods for declaring one-dimensional arrays of integer or symbolic variables are:

```
Var[] array(String id, Size1D size, IntToDomInteger f, String note)
Var[] array(String id, Size1D size, IntToDomInteger f)
Var[] array(String id, Size1D size, XDomInteger dom, String note)
Var[] array(String id, Size1D size, XDomInteger dom)

VarSymbolic[] arraySymbolic(String id, Size1D size, IntToDomSymbolic f, String note)
VarSymbolic[] arraySymbolic(String id, Size1D size, IntToDomSymbolic f)
VarSymbolic[] arraySymbolic(String id, Size1D size, XDomSymbolic dom, String note)
VarSymbolic[] arraySymbolic(String id, Size1D size, XDomSymbolic dom)
```

where all methods require an object `Size1D` that can be simply obtained by calling the following method:

```
Size1D size(int length)
```

Some of these methods accept lambda functions in order to let the user the possibility of declaring in the same array variables with different domains. For example, suppose that we have analytically deduced that the two first variables of the array must be assigned a value strictly less than 6 and the two last variables of the array must be assigned a value strictly less than 9. We can write:

```
class Riddle3 implements ProblemAPI {

  public void model() {
    Var x[] = array("x", size(4), i -> dom(range(i < 2 ? 6 : 9)));

    equal(add(x[0], 1), x[1]);
    equal(add(x[1], 1), x[2]);
    equal(add(x[2], 1), x[3]);
    equal(add(x[0], x[1], x[2], x[3]), 14);
  }
}
```

and the XCSP3 file obtained after compilation is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" size="[4]">
      <domain for="x[0] x[1]"> 0..5 </domain>
      <domain for="x[2] x[3]"> 0..8 </domain>
    </array>
  </variables>
  <constraints>
    <intension> eq(add(x[0],1),x[1]) </intension>
    <intension> eq(add(x[1],1),x[2]) </intension>
    <intension> eq(add(x[2],1),x[3]) </intension>
    <intension> eq(add(x[0],x[1],x[2],x[3]),14) </intension>
  </constraints>
</instance>
```

Because the three binary constraints are similar, one may wonder if we couldn't post these constraints in a group (loop). This is indeed possible by using the following method:

```
CtrArray forall(Range range, IntConsumer c)
```

The behaviour is as follows: the specified consumer object is called for each value in the specified range, and most of the time the goal is to post a constraint when it is called. Let us see this in action:

```
class Riddle4 implements ProblemAPI {

  public void model() {
    Var x[] = array("x", size(4), dom(range(15)));

    forall(range(3), i -> equal(add(x[i], 1), x[i + 1]));

    equal(add(x[0], x[1], x[2], x[3]), 14);
  }
}
```

and the XCSP3 file obtained after compilation is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" size="[4]"> 0..14 </array>
  </variables>
  <constraints>
    <group>
      <intension> eq(add(%0,%1),%2) </intension>
      <args> x[0] 1 x[1] </args>
      <args> x[1] 1 x[2] </args>
      <args> x[2] 1 x[3] </args>
    </group>
    <intension> eq(add(x[0],x[1],x[2],x[3]),14) </intension>
  </constraints>
</instance>
```

Finally, it seems more appropriate to represent the last constraint as a constraint `sum`. There are many ways to post a constraint `sum`, as you can see in the Javadoc of the class `ProblemAPI`. This gives:

```
class Riddle5 implements ProblemAPI {

  public void model() {
    Var x[] = array("x", size(4), dom(range(15)));

    forall(range(3), i -> equal(add(x[i], 1), x[i + 1]));
    sum(x, EQ, 14);
  }
}
```

and the XCSP3 file obtained after compilation is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" size="[4]"> 0..14 </array>
  </variables>
  <constraints>
    <group>
      <intension> eq(add(%0,%1),%2) </intension>
      <args> x[0] 1 x[1] </args>
      <args> x[1] 1 x[2] </args>
      <args> x[2] 1 x[3] </args>
    </group>
    <sum>
      <list> x[] </list>
      <condition> (eq,14) </condition>
    </sum>
  </constraints>
</instance>
```

2.2 Playing with Small Constraint Networks

When studying properties of constraint networks, it is frequent to draw some small constraint networks under the form of compatibility graphs (also called microstructure). For example, Figure 3 presents the compatibility graph of a small constraint network P such that:

- the set of variables of P is $vars(P) = \{x, y, z\}$, each variable having $\{a, b\}$ as domain;
- the set of constraints of P is $ctrs(P) = \{\langle x, y\rangle \in \{(a, a), (b, b)\}, \langle x, z\rangle \in \{(a, a), (b, b)\}, \langle y, z\rangle \in \{(a, b), (b, a)\}$.

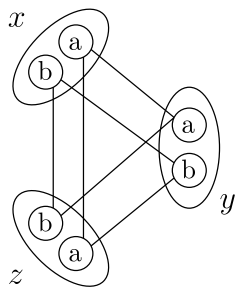


Figure 3: The compatibility graph of a small constraint network.

The interested reader can observe that the constraint network is arc-consistent (AC) but not path-inverse consistent (PIC). Anyway, the MCSP3 model is as follows:

```
public class Toy implements ProblemAPI {

  public void model() {
    VarSymbolic x = var("x", dom("a", "b"));
    VarSymbolic y = var("y", dom("a", "b"));
    VarSymbolic z = var("z", dom("a", "b"));

    extension(vars(x, y), table("(a,a)(b,b)"));
    extension(vars(x, z), table("(a,a)(b,b)"));
    extension(vars(y, z), table("(a,b)(b,a)"));
  }
}
```

and the XCSP3 file obtained after compilation is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <var id="x" type="symbolic"> a b </var>
    <var id="y" type="symbolic"> a b </var>
    <var id="z" type="symbolic"> a b </var>
  </variables>
  <constraints>
    <extension>
      <list> x y </list>
      <supports> (a,a)(b,b) </supports>
    </extension>
    <extension>
      <list> x z </list>
      <supports> (a,a)(b,b) </supports>
    </extension>
    <extension>
      <list> y z </list>
      <supports> (a,b)(b,a) </supports>
    </extension>
  </constraints>
</instance>
```

Here, we declare three stand-alone symbolic variables (note how the domain of each of them is simply composed of the two symbols "a" and "b"). And we declare three binary constraints `extension`. First, note that the scope of the constraints is specified using the method `vars()`. The many overloading versions of `vars()` permit to build 1-dimensional arrays of variables from a sequence of parameters, where each element of the sequence must only contain variables (and possibly null values), either stand-alone or present in arrays (of any dimension). All variables are collected in order, and concatenated to form a 1-dimensional array (note that null values are simply discarded).

Below, you can find all the methods that can be used to post extensional constraints.

```
CtrAlone extension(Var[] scp, int[][] tuples, Boolean positive)
CtrAlone extension(Var[] scp, int[]... tuples)
CtrAlone extension(Var[] scp, Table table)
CtrAlone extension(Var x, int[] values, Boolean positive)
CtrAlone extension(Var x, int... values)

CtrAlone extension(VarSymbolic[] scp, String[][] tuples, Boolean positive)
CtrAlone extension(VarSymbolic[] scp, String[]... tuples)
CtrAlone extension(VarSymbolic[] scp, Table table)
CtrAlone extension(VarSymbolic x, String[] values, Boolean positive)
CtrAlone extension(VarSymbolic x, String... values)
```

When the parameter `positive` is present, it indicates if the constraint lists the allowed tuples (value *true*) of the forbidden tuples (value *false*). Note that it is possible to post unary constraints and non-unary constraints. Specifying tuples is made possible under the form of arrays or `Table` objects. Below, here are the methods for building `Table` objects:

```
Table table()
Table table(String tuples)

TableInteger table(int[] tuple)
TableInteger table(int val1, int... otherVals)
TableInteger table(int[] tuple1, int[]... otherTuples)

TableSymbolic table(String[] tuple)
TableSymbolic table(String val1, String... otherVals)
TableSymbolic table(String[] tuple1, String[]... otherTuples)
```

Instead of:

```
extension(vars(x, y), table("(a,a)(b,b)"));
```

we could have equivalently written:

```
extension(vars(x, y), new String[][] {{"a", "a"},{"b", "b"}});
```

Note that there are several methods in classes `Table`, `TableInteger` and `TableSymbolic` to build tables in a few steps. You also indicate if the table is positive or negative.

If instead of declaring symbolic variables, you prefer to declare integer variables, replacing "a" by 0 and "b" by 1, you have to write:

```
public class Toy implements ProblemAPI {

  public void model() {
    Var x = var("x", dom(0, 1));
    Var y = var("y", dom(0, 1));
    Var z = var("z", dom(0, 1));

    extension(vars(x, y), table("(0,0)(1,1)"));
    extension(vars(x, z), table("(0,0)(1,1)"));
    extension(vars(y, z), table("(0,1)(1,0)"));
  }
}
```

Again, instead of:

```
extension(vars(x, y), table("(0,0)(1,1)"));
```

we could have equivalently written:

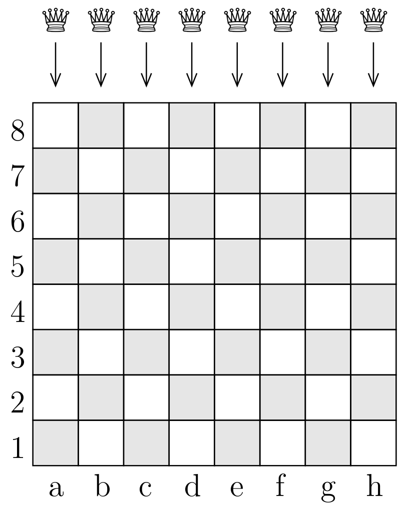
```
extension(vars(x, y), new int[][] {{0, 0},{1, 1}});
```

3 Academic Problems

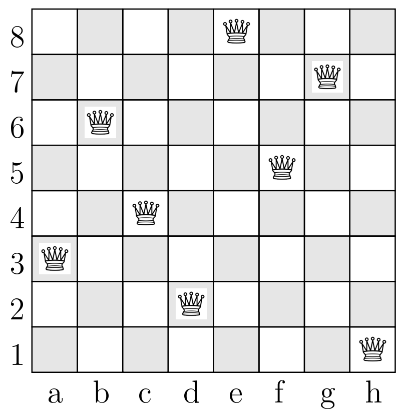
Contrary to single problems, academic problems require the introduction of some pieces of data by the user.

3.1 Queens

The problem is stated as follows: can we put 8 queens on a chessboard such that no two queens attack each other? Two queens attack each other iff they belong to the same row, the same column or the same diagonal. An illustration is given by Figure 4.



(a) Puzzle



(b) Solution

Figure 4: Putting 8 queens on a chessboard

By considering boards of various size, the problem can be generalized as follows: can we put n queens on a board of size $n \times n$ such that no two queens attack each other? Contrary to previously introduced single problems, we have to deal here with a family of problem instances, each of them characterized by a distinct value of n. We can try to solve the 8-queens instance, the 10-queens instance, and even the 1000-queens instance.

For such problems, we have to separate the description of the model from the description of the data. In other words, we have to write a model with some kind of parameters. In MCSP3, this is quite natural and easy to do:

1. clearly identify the parameters of the problem (with their types)
2. introduce fields representing them in the class implementing `ProblemAPI`
3. specify values for these parameters (fields) when you compile to XCSP3

In our case, we have only one integer parameter called n. If we associate a variable q_i with the ith row of the board, then we can simply post the following constraints:

$$q_i \neq q_j \wedge |q_i - q_j| \neq |i - j|, \forall i, j : 1 \leq i < j \leq n$$

This is translated as:

```
class Queens implements ProblemAPI {
  int n; // number of queens

  public void model() {
    Var[] q = array("q", size(n), dom(range(n)));

    forall(range(n).range(n), (i, j) −> {
      if (i < j)
        intension(and(ne(q[i], q[j]), ne(dist(i, j), dist(q[i], q[j]))));
    });
  }
}
```

Note how the parameter n is used in Method `model()`. To post the constraints `intension`, we use the following method:

```
CtrArray forall(Rangesx2 rangesx2, Intx2Consumer c2)
```

An object `Rangesx2` represents the Cartesian product of two basic ranges. In our case, it is obtained by:

```
range(n).range(n)
```

So, when we write:

```
forall(range(n).range(n), (i, j) -> ...
```

it means: $\forall (i, j) \in 0..n-1 \times 0..n-1, \ldots$

Now, the question is: how can we solve a specific instance? The answer is: just compile the model while indicating with the argument `-data=` either the value for n or the name of a JSON file containing an object with a unique field n. In the former case, this gives for $n = 4$:

```
java org.xcsp.modeler.Compiler Queens -data=4
```

and the XCSP3 file obtained after compilation is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="q" size="[4]"> 0..3 </array>
  </variables>
  <constraints>
    <group>
      <intension> and(ne(%0,%1),ne(%2,dist(%0,%1))) </intension>
      <args> q[0] q[1] 1 </args>
      <args> q[0] q[2] 2 </args>
      <args> q[0] q[3] 3 </args>
      <args> q[1] q[2] 1 </args>
      <args> q[1] q[3] 2 </args>
      <args> q[2] q[3] 1 </args>
    </group>
  </constraints>
</instance>
```

In the latter case, just build a file “queens-4.json” whose content is:

```
{
  "n": 4
}
```

and execute:

```
java org.xcsp.modeler.Compiler Queens -data=queens-4.json
```

At this point, you have been told that it could be a good idea to post a constraint `allDifferent`. So, you would like to test a slightly different model, and you think that it is annoying of multiplying files. Actually, you can put different models in the same file by using the method `isModel` that accepts a string as parameter. When you compile, you have then to indicate the name of the model. Putting two slightly different models in the same file gives:

```
class Queens implements ProblemAPI {
  int n; // number of queens

  public void model() {
    Var[] q = array("q", size(n), dom(range(n)));

    if (isModel("m1")) {
      forall(range(n).range(n), (i, j) −> {
        if (i < j)
          intension(and(ne(q[i], q[j]), ne(dist(i, j), dist(q[i], q[j]))));
      });
    }
    if (isModel("m2")) {
      allDifferent(q);
      forall(range(n).range(n), (i, j) -> {
        if (i < j)
          notEqual(dist(i, j), dist(q[i], q[j]));
      });
    }
  }
}
```

To compile the first model ("m1"), just type:

```
java org.xcsp.modeler.Compiler Queens -data=8 -model=m1
```

To compile the second model ("m2"), just type:

```
java org.xcsp.modeler.Compiler Queens -data=8 -model=m2
```

3.2 Board Coloration Problem

The (chess)board coloration problem is to color all squares of a board composed of r rows and c colomns such that the four corners of any rectangle in the board must not be assigned the same color. Importantly, we want to minimize the number of used colors.

Figure 5: Coloring Boards.

This time, we need two integer parameters r and c. After a very rough analysis, we can decide to use $r \times c$ as an upper bound of the number of used colors. This gives:

```
class BoardColoration implements ProblemAPI {
  int r; // number of rows
  int c; // number of columns

  public void model() {
    Var[][] x = array("x", size(r, c), dom(range(r * c)));

    forall(range(r).range(r).range(c).range(c), (i1, i2, j1, j2) -> {
      if (i1 < i2 && j1 < j2)
      notAllEqual(x[i1][j1], x[i1][j2], x[i2][j1], x[i2][j2]);
    });

    minimize(MAXIMUM, x);
  }
}
```

Here, we declare a two-dimensional array of variables: its id is “x”, its size is $r \times c$ and each of its variables has $\{0, 1, \ldots, r \times c - 1\}$ as domain. Note that indexing starts at 0. The methods for declaring two-dimensional arrays of integer variables are:

```
Var[][] array(String id, Size2D size, Intx2ToDomInteger f, String note)
Var[][] array(String id, Size2D size, Intx2ToDomInteger f)
Var[][] array(String id, Size2D size, XDomInteger dom, String note)
Var[][] array(String id, Size2D size, XDomInteger dom)
```

where all methods require an object `Size2D` that can be simply obtained by calling the following method:

```
Size2D size(int length1, int length2)
```

To post the constraints `notAllEqual`, we use the following method:

```
CtrArray forall(Rangesx4 rangesx4, Intx4Consumer c4)
```

An object `Rangesx4` represents the Cartesian product of four basic ranges. In our case, it is obtained by:

```
range(r).range(r).range(c).range(c)
```

So, when we write:

```
forall(range(r).range(r).range(c).range(c), (i1, i2, j1, j2) -> ...
```

it means: $\forall(i1, i2, j1, j2) \in 0..r-1 \times 0..r-1 \times 0..c-1 \times 0..c-1, \ldots$

Finally, the objective function corresponds to the minimization of the maximum value taken by any variable in the two-dimensional array x. Because domains are all similar, this is indeed equivalent to minimize the number of used colors. Here are the methods that can be called to post an objective function (when no coefficients are required):

```
ObjEntity minimize(IVar x)
ObjEntity minimize(TypeObjective type, IVar... list)
ObjEntity minimize(TypeObjective type, IVar[][] list)
ObjEntity minimize(TypeObjective type, IVar[][][] list)

ObjEntity maximize(IVar x)
ObjEntity maximize(TypeObjective type, IVar... list)
ObjEntity maximize(TypeObjective type, IVar[][] list)
ObjEntity maximize(TypeObjective type, IVar[][][] list)
```

The sepcified type must take one of the following value[3]:

- EXPRESSION
- SUM
- PRODUCT
- MINIMUM
- MAXIMUM
- NVALUES
- LEX

To solve a specific instance, as usually, we have first to compile the model while indicating with the argument `-data=` either the values for r and c or the name of a JSON file containing an object with two fields r and c. In the former case, this gives for $r = 3$ and $c = 3$:

```
java org.xcsp.modeler.Compiler BoardColoration -data=[3,3]
```

As you can observe, you need to use square brackets to surround values sepated by commas (no whitespace authorized). The XCSP3 file obtained after compilation is:

```
<instance format="XCSP3" type="COP">
  <variables>
    <array id="x" size="[3][3]"> 0..8 </array>
  </variables>
  <constraints>
    <group>
      <nValues>
        <list> %... </list>
        <condition> (gt,1) </condition>
      </nValues>
      <args> x[0][0] x[0][1] x[1][0] x[1][1] </args>
      <args> x[0][0] x[0][2] x[1][0] x[1][2] </args>
      <args> x[0][1] x[0][2] x[1][1] x[1][2] </args>
      <args> x[0][0] x[0][1] x[2][0] x[2][1] </args>
      <args> x[0][0] x[0][2] x[2][0] x[2][2] </args>
      <args> x[0][1] x[0][2] x[2][1] x[2][2] </args>
      <args> x[1][0] x[1][1] x[2][0] x[2]]1] </args>
      <args> x[1][0] x[1][2] x[2][0] x[2][2] </args>
      <args> x[1][1] x[1][2] x[2][1] x[2][2] </args>
    </group>
  </constraints>
  <objectives>
    <minimize type="maximum"> x[][] </minimize>
  </objectives>
</instance>
```

[3]See Chapter 3 in XCSP3 Specifications for more information.

We can also build a file “board-3-3.json” whose content is:

```
{
  "r": 3,
  "c": 3
}
```

and execute:

```
java org.xcsp.modeler.Compiler BoardColoration -data=board-3-3.json
```

3.3 Magic Sequence

A magic sequence of order n is a sequence of integers $x_0, \ldots, x_{n-1}$ between 0 and $n-1$, such that each value $i \in 0..n1$ occurs exactly x_i times in the sequence. For example,

```
6 2 1 0 0 0 1 0 0 0
```

is a magic sequence of order 10 since 0 occurs 6 times, 1 occurs twice, ... and 9 occurs 0 times.

One can prove that every solution respects:

$$x_0 + x_1 + x_2 + x_3 + \cdots + x_{n-1} = 0$$

and

$$-1x_0 + 0x_1 + 1x_2 + 2x_3 + \cdots + (n-2)x_{n-1} = 0$$

So, it may be a good idea to post these additional constraints and making it clear that they are redundant (i.e., not modifying the set of solutions). This gives:

```
class MagicSequence implements ProblemAPI {
  int n;

  public void model() {
    Var[] x = array("x", size(n), dom(range(n)));

    cardinality(x, range(n), occurrences(x));

    block(() -> {
      sum(x, EQ, n);
      sum(x, range(-1, n - 2), EQ, 0);
    }).tag(REDUNDANT_CONSTRAINTS);
  }
}
```

On the one hand, the constraint `cardinality` is exactly what we need here. We invite the reader to look at the overloaded methods for `cardinality` in Class `ProblemAPI` and see how we can manage occurrences through many methods. On the other hand, we have put together the two additional constraints in a so-called block, permitting to tag these two constraints (actually, the block) with the information “redundantConstraints”. The method that we have used is:

```
CtrArray block(Runnable r)
```

and at this point, it is interesting to note that we can tag any constraint or group of constraints by calling `tag()`. If we execute:

```
java org.xcsp.modeler.Compiler MagicSequence -data=6
```

we obtain the following XCSP3 instance:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" size="[6]"> 0..5 </array>
  </variables>
  <constraints>
    <cardinality>
      <list> x[] </list>
      <values> 0 1 2 3 4 5 </values>
      <occurs> x[] </occurs>
    </cardinality>
    <block class="redundantConstraints">
      <sum>
        <list> x[] </list>
        <condition> (eq,6) </condition>
      </sum>
      <sum>
        <list> x[] </list>
        <coeffs> −1 0 1 2 3 4 </coeffs>
        <condition> (eq,0) </condition>
      </sum>
    </block>
  </constraints>
</instance>
```

Note the presence of the attribute `class` that results from the call to method `tag()`.

4 Structured Problems

Some problems need more than elementary data. We call them structured problems for simplicity here.

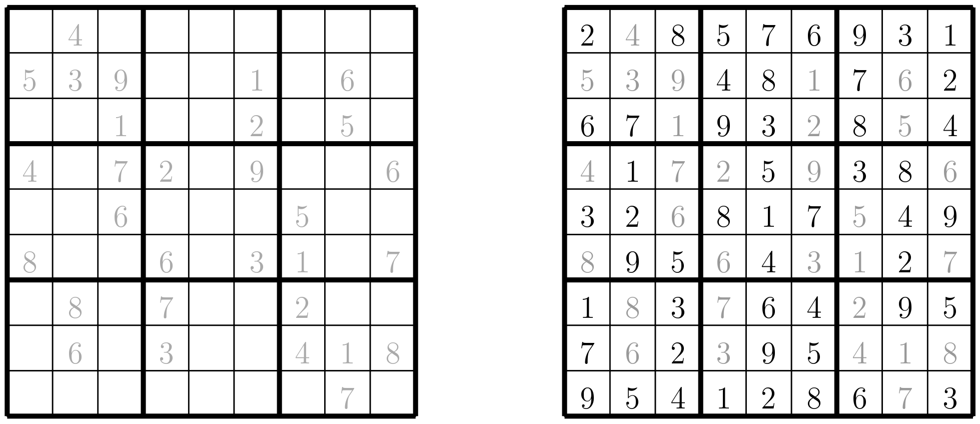
4.1 Sudoku

This well-known problem is stated as follows: fill in a grid using digits ranging from 1 to 9 such that:

- all digits occur on each row
- all digits occur on each column

- all digits occur in each 3×3 block (starting at a position multiple of 1)

An illustration is given by Figure 6.



(a) Puzzle (b) Solution

Figure 6: Solving a Sudoku Grid

Because there are several clues, and because their number cannot be anticipated, we need a parameter *grid* that represents a two-dimensional array of integer values. When $grid[i][j]$ is 0, it means that the cell is empty, whereas when it contains a digit between 1 and 9, it means that it represents a fixed value (clue). The model we can write is then:

```
class Sudoku implements ProblemAPI {
  int[][] grid; // if not 0, grid[i][j] is a value imposed at row i and col j

  public void model() {
    Var[][] x = array("x", size(9, 9), dom(range(1, 9)));

    allDifferentMatrix(x);
    forall(range(0, 8, 3).range(0, 8, 3), (i, j) −> {
      Var[] t = select(x, range(i, i + 2).range(j, j + 2));
      allDifferent(t);
    }).tag(BLOCKS);

    instantiation(x, grid, (i, j) −> grid[i][j] != 0).tag(CLUES);
  }
}
```

To ensure that all digits at different on each row and each column, we use:

```
CtrEntity allDifferentMatrix(Var[][] matrix)
```

Then, we have to post a constraint `allDifferent` on each block. Note how we use the following method `select()`:

```
<T extends IVar> T[] select(T[][] vars, Rangesx2 rangesx2)
```

to extract the right subset of 9 variables for each block.

Finally, each clue (given value) represents a unary constraint. We can post them using the constraint `instantiation`. The method that we use here is:

```
CtrEntity instantiation(Var[][] list, int[][] values, Intx2Predicate p)
```

that permits to assign the variable $list[i][j]$ with the value $values[i][j]$ provided that p returns true for (i, j).

Suppose now that we have a file 'grid.json" containing:

```
{
  "grid": [
    [0,4,0,0,0,0,0,0,0],
    [5,3,9,0,0,1,0,6,0],
    [0,0,1,0,0,2,0,5,0],
    [4,0,7,2,0,9,0,0,6],
    [0,0,6,0,0,0,5,0,0],
    [8,0,0,6,0,3,1,0,7],
    [0,8,0,7,0,0,2,0,0],
    [0,6,0,3,0,0,4,1,8],
    [0,0,0,0,0,0,0,7,0]
  ]
}
```

then, we can execeute:

```
java org.xcsp.modeler.Compiler Sudoku -data=grid.json
```

and we obtain the following (simplified here) XCSP3 instance:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" size="[9][9]"> 1..9 </array>
  </variables>
  <constraints>
    <allDifferent>
      <matrix> x[][] </matrix>
    </allDifferent>
    <group>
      <allDifferent> %... </allDifferent>
      <args> x[0..2][0..2] </args>
      <args> x[0..2][3..5] </args>
      <args> x[0..2][6..8] </args>
      <args> x[3..5][0..2] </args>
      <args> x[3..5][3..5] </args>
      <args> x[3..5][6..8] </args>
      <args> x[6..8][0..2] </args>
      <args> x[6..8][3..5] </args>
      <args> x[6..8][6..8] </args>
    </group>
    <instantiation class="clues" note="Just 2 clues here for the simplicity of the illustration">
      <list> x[0][1] x[8][7] </list>
      <values> 4 7 </values>
    </instantiation>
  </constraints>
</instance>
```

4.2 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 CSPLib–Problem 034.

Figure 7: Warehouse.

An example of data is the file "warehouse.json" containing:

```
{
  "fixedCost": 30,
  "warehouseCapacities": [1,4,2,1,3],
  "storeSupplyCosts": [
    [100,24,11,25,30],[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]
  ]
}
```

A possible model is:

```
class Warehouse implements ProblemAPI {
  int fixedCost;
  int[] warehouseCapacities;
  int[][] storeSupplyCosts;

  public void model() {
    int nWarehouses = warehouseCapacities.length;
    int nStores = storeSupplyCosts.length;

    Var[] s = array("s", size(nStores), dom(range(nWarehouses)),
      "s[i] is the warehouse supplier of store i");
    Var[] c = array("c", size(nStores), i -> dom(storeSupplyCosts[i]),
      "c[i] is the cost of supplying store i");
    Var[] o = array("o", size(nWarehouses), dom(0, 1),
      "o[i] is 1 if the warehouse i is open");

    forall(range(nWarehouses),
      i -> atMost(s, i, warehouseCapacities[i]));
    forall(range(nStores),
      i -> element(o, s[i], 1));
    forall(range(nStores),
      i -> element(storeSupplyCosts[i], s[i], c[i]));

    int[] coeffs = vals(repeat(1,nStores),repeat(fixedCost,nWarehouses));
    minimize(SUM, vars(c,o), coeffs);
  }
}
```

Here, it is intersting to see how we can define a specific domain to each variable of the array c by means of a simple lambda function. Remember that values are sorted and made distinct. The objective function corresponds to minimizing a weighted sum. Note how `vars(c,o)` builds a 1-dimensional array from two arrays, and how similarly the method `vals()` concatenates integer values from two arrays to form a 1-dimensional array of integers. On our example, the array $coeffs$ is `1 1 1 1 1 1 1 1 1 1 30 30 30 30 30` because we have 10 stores and 5 warehouses.

After executing:

```
java org.xcsp.modeler.Compiler Warehouse -data=warehouse.json
```

and we obtain the following XCSP3 instance (constraints are ommitted):

```
<instance format="XCSP3" type="COP">
  <variables>
    <array id="s" note="s[i] is the warehouse supplier of store i" size="[10]"> 0..4 </array>
    <array id="c" note="c[i] is the cost of supplying store i" size="[10]">
      <domain for="c[0]"> 11 24 25 30 100 </domain>
      <domain for="c[1]"> 27 28 74 82 83 </domain>
      <domain for="c[2]"> 70 71 74 96 97 </domain>
      <domain for="c[3]"> 2 55 61 69 73 </domain>
    <domain for="c[4]"> 4 46 59 83 96 </domain>
    <domain for="c[5]"> 22 29 42 59 67 </domain>
    <domain for="c[6]"> 1 5 56 59 73 </domain>
    <domain for="c[7]"> 10 13 43 73 96 </domain>
    <domain for="c[8]"> 35 46 63 85 93 </domain>
    <domain for="c[9]"> 47 55 65 71 95 </domain>
   </array>
   <array id="o" note="o[i] is 1 if the warehouse i is open" size="[5]"> 0 1 </array>
  </variables>
  <constraints>
   ...
  </constraints>
  <objectives>
   <minimize type="sum">
    <list> c[] o[] </list>
    <coeffs> 1 1 1 1 1 1 1 1 1 1 30 30 30 30 30 </coeffs>
   </minimize>
  </objectives>
</instance>
```

4.3 Blackhole

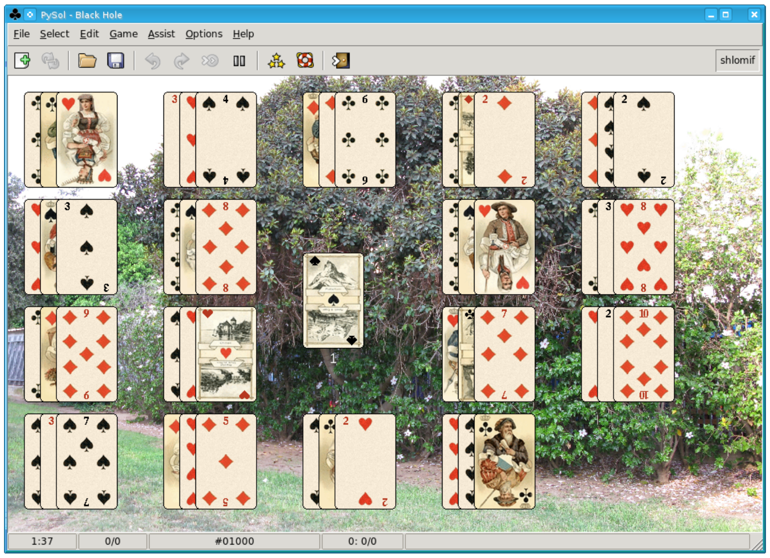


Figure 8: Blackhole.

From WikiPedia. "Black Hole is a solitaire card game. Invented by David Parlett, this game's objective is to compress the entire deck into one foundation. The cards are dealt to a board in piles of three. The leftover card, dealt first or last, is placed as a single foundation called the Black Hole. This card usually is the Ace of Spades. Only the top cards of each pile in the tableau are available for play and in order for a card to be placed in the Black Hole, it must be a rank higher or lower than the top card on the Black Hole. This is the only allowable move in the entire game. The game ends if there are no more top cards that can be moved to the Black Hole. The game is won if all of the cards end up in the Black Hole." An illustration is given by Figure 8.

We may want to play with various sizes of piles and various number of cards per suit. An example of data is the file "blackhole.json" containing:

```
{
  "nCardsPerSuit": 4,
  "nCardsPerPile": 3,
  "piles": [[1,4,13],[15,9,6],[14,2,12],[7,8,5],[11,10,3]]}
}
```

A possible model is:

```
class Blackhole implements ProblemAPI {

  int nCardsPerSuit, nCardsPerPile;
  int[][] piles;

  public void model() {
    int nCards = 4 * nCardsPerSuit;
    int nPiles = (nCards - 1) / nCardsPerPile;

    Var[] x = array("x", size(nCards), dom(range(nCards)),
      "x[i] is the value j of the card at the ith position of the built stack.");
    Var[] y = array("y", size(nCards), dom(range(nCards)),
      "y[j] is the position i of the card whose value is j");

    channel(x, y);
    equal(y[0], 0).note("The ace od Spades is the card initially on the stack");

    forall(range(nPiles).range(nCardsPerPile - 1),
      (i, j) -> lessThan(y[piles[i][j]], y[piles[i][j + 1]])
    ).note("Cards must be played in the order of the piles");

    int[][] tuples = range(nCards).range(nCards).select(
      (i, j) -> i % nCardsPerSuit == (j + 1) % nCardsPerSuit ||
                j % nCardsPerSuit == (i + 1) % nCardsPerSuit);
    slide(x, range(nCards - 1), i -> extension(vars(x[i], x[i + 1]), tuples));
  }
}
```

Note how the constraint `channel` is used to make a channelling between the two arrays x and y, and how tuples are built using an object `Rangesx2` and a lambda function given as parameter of the method `select()`. Finally, we use the meta-constraint `slide` to guarantee that each new card is at a rank higher or lower than the previous one.

After executing:

```
java org.xcsp.modeler.Compiler Blackhole -data=blackhole.json
```

and we obtain the following XCSP3 instance (some tuples have been ommitted; see the ellipsis . . .):

```
<instance format="XCSP3" type="CSP">
  <variables>
  <array id="x" note="x[i] is the value j of the card at the ith pos. of the stack." size="[16]">
    0..15
  </array>
  <array id="y" note="y[j] is the position i of the card whose value is j" size="[16]">
    0..15
  </array>
  </variables>
  <constraints>
    <channel>
      <list> x[] </list>
      <list> y[] </list>
    </channel>
    <intension note="The ace od Spades is initially on the stack"> eq(y[0],0) </intension>
    <group note="Cards must be played in the order of the piles">
      <intension> lt(%0,%1) </intension>
      <args> y[1] y[4] </args>
      <args> y[4] y[13] </args>
      <args> y[15] y[9] </args>
      <args> y[9] y[6] </args>
      <args> y[14] y[2] </args>
      <args> y[2] y[12] </args>
      <args> y[7] y[8] </args>
      <args> y[8] y[5] </args>
      <args> y[11] y[10] </args>
      <args> y[10] y[3] </args>
    </group>
    <slide>
      <list collect="2"> x[] </list>
      <extension>
        <list> %0 %1 </list>
        <supports> (0,1)(0,3)(0,5)(0,7)(0,9)(0,11)...(15,14) </supports>
      </extension>
    </slide>
  </constraints>
</instance>
```

4.4 Rack Configuration Problem

The rack configuration problem consists of plugging a set of electronic cards into racks with electronic connectors. Each card plugged into a rack uses a connector. In order to plug a card into a rack, the rack must be of a rack model. Each card is characterised by the power it requires. Each rack model is characterised by the maximal power it can supply, its number of connectors, and its price. The problem is to decide how many of the available racks are actually needed such that:

- every card is plugged into one rack
- the total power demand and the number of connectors required by the cards does not exceed that available for a rack
- the total price is minimised.

See CSPLib–Problem 031.

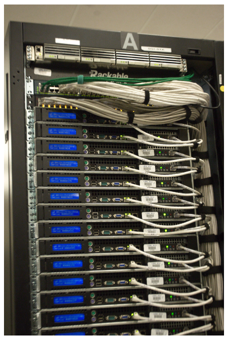


Figure 9: Rack.

An example of data is the file "rack.json" containing:

```
{
  "nRacks": 10,
  "models": [[150,8,150],[200,16,200]],
  "cardTypes": [[20,20],[40,8],[50,4],[75,2]]
}
```

A possible model is:

```
class Rack implements ProblemAPI {
  int nRacks;
  int[][] models;
  int[][] cardTypes;

  public void model() {
    models = addObject(models, tuple(0, 0, 0), 0); // we add first a dummy model (0,0,0)
    int nModels = models.length;
    int nTypes = cardTypes.length;
    int[] powers = columnOf(models, 0);
    int[] connectors = columnOf(models, 1);
    int[] prices = columnOf(models, 2);
    int[] cardPowers = columnOf(cardTypes, 0);
    int maxCapacity = IntStream.of(connectors).max().orElse(-1);

    Var[] r = array("r", size(nRacks), dom(range(nModels)),
      "r[i] is the model used for the ith rack");
    Var[][] c = array("c", size(nRacks, nTypes), (i, j) -> dom(range(0, Math.min(maxCapacity,
         cardTypes[j][1]))), "c[i][j] is the number of cards of type j put in the ith rack");
    Var[] rpw = array("rpw", size(nRacks), dom(powers),
      "rpw[i] is the power of the ith rack");
    Var[] rcn = array("rcn", size(nRacks), dom(connectors),
      "rcn[i] is the number of connectors of the ith rack");
    Var[] rpr = array("rpr", size(nRacks), dom(prices),
      "rpr[i] is the price of the ith rack");

    forall(range(nRacks), i -> extension(vars(r[i], rpw[i]), number(powers))).
      note("linking the ith rack with its power");
    forall(range(nRacks), i -> extension(vars(r[i], rcn[i]), number(connectors))).
      note("linking the ith rack with its number of connectors");
    forall(range(nRacks), i -> extension(vars(r[i], rpr[i]), number(prices))).
      note("linking the ith rack with its price");
    forall(range(nRacks), i -> sum(c[i], LE, rcn[i])).
      note("connector-capacity constraints");
    forall(range(nRacks), i -> sum(c[i], cardPowers, LE, rpw[i])).
      note("power-capacity constraints");
    forall(range(nTypes), i -> sum(columnOf(c, i), EQ, cardTypes[i][1])).
      note("demand constraints");
    decreasing(r).tag(SYMMETRY_BREAKING);

    minimize(SUM, rpr);
  }
}
```

Note how we use the method `columnOf()` for getting the ith column of a two-dimensional array and the method `number()` for converting a one-dimensional array into a two-dimensional array. For example, if an array t is defined by `int[] t = {2,5,1}` then `number(t)` returns an array m just as if m would have been defined by `int[][] m = {{0,2},{1,5},{2,1}}`.

One drawback with the previous model is that it is difficult to understand the role of each piece of data, when we look at the JSON file isolated. One possibility would be to propose a better structure as in this file "rackb.json":

```
{
  "nRacks": 10,
  "rackModels": [
    {"power":150,"nConnectors":8,"price":150},
    {"power":200,"nConnectors":16,"price":200}
  ],
  "cardTypes": [
    {"power":20,"demand":20},
    {"power":40,"demand":8},
    {"power":50,"demand":4},
    {"power":75,"demand":2}
  ]
}
```

Actually, in XCSP3, we can easily deal with such structured data, by developping a class for each type of objects. This is what we do below with two inner classes, allowing us to represent rack models and card types. The classes must be public, the fields must be public and we must have exactly one public constrctor.

```
class Rack2 implements ProblemAPI {
  int nRacks;
  RackModel[] rackModels;
  CardType[] cardTypes;

  public class RackModel {
    public int power, nConnectors, price;

    public RackModel(int power, int nConnectors, int price) {
      this.power = power;
      this.nConnectors = nConnectors;
      this.price = price;
    }
  }

  public class CardType {
    public int power, demand;

    public CardType(int power, int demand) {
      this.power = power;
      this.demand = demand;
    }
  }

  public void model() {
    rackModels = addObject(rackModels, new RackModel(0, 0, 0), 0); // we add a dummy model
    int nModels = rackModels.length;
    int nTypes = cardTypes.length;
    int[] powers = Stream.of(rackModels).mapToInt(r -> r.power).toArray();
    int[] connectors = Stream.of(rackModels).mapToInt(r -> r.nConnectors).toArray();
    int[] prices = Stream.of(rackModels).mapToInt(r -> r.price).toArray();
    int[] cardPowers = Stream.of(cardTypes).mapToInt(r -> r.power).toArray();
    int maxCapacity = IntStream.of(connectors).max().orElse(-1);

    ... // the rest of the model remains the same
  }
}
```