PyCSP^3
Modeling Combinatorial Constrained Problems in Python

Christophe Lecoutre and Nicolas Szczepanski
University of Artois
CRIL CNRS, UMR 8188
France

{lecoutre,szczepanski}@cril.fr

December 24, 2019

Abstract

In this document, we introduce PyCSP^3, a Python library that allows us to write models of combinatorial constrained problems in a natural and declarative way. Currently, with PyCSP^3, you can write models of satisfaction and optimization problems. More specifically, you can build CSP (Constraint Satisfaction Problem) and COP (Constraint Optimization Problem) models. Importantly, there is a complete separation between modeling and solving phases: you write a model, you compile it (while providing some data) in order to obtain an XCSP^3 instance (file), and you solve that problem instance with a constraint solver. PyCSP^3 is inspired from both JvCSP^3 [12] and Numberjack [8]. It has also connections with CPpy [7].

The main ingredients of the complete tool chain we propose for solving combinatorial constrained problems are:

- PyCSP^3: a Python library for modeling constrained problems, which is described in this document (or equivalently, JvCSP^3, a Java-based API)
- XCSP^3: an intermediate format used to represent problem instances while preserving structure of models

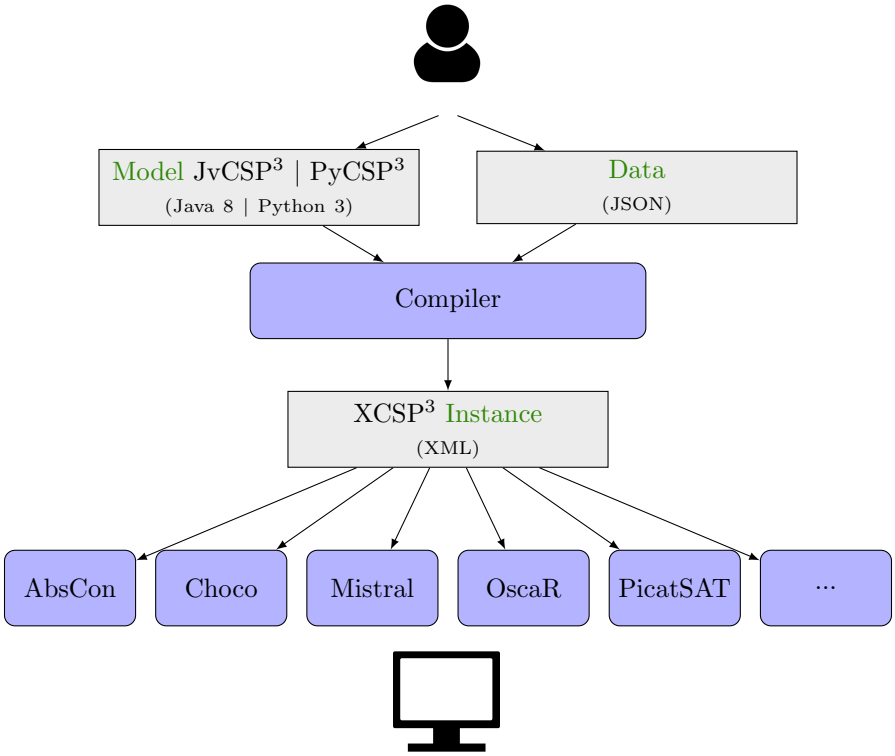


Figure 1: Complete process for modeling and solving combinatorial constrained problems.

For modeling, as indicated above, the user can choose between two well-known languages (Python and Java), but this document focuses on Python. As shown in Figure 1, the user who wishes to solve a combinatorial constrained problem has to:

1. write a model using either the Python library PyCSP^3 (i.e., write a Python file) or the Java modeling API JvCSP^3 (i.e., write a Java file)
2. provide a data file (in JSON format) for a specific problem instance to be solved
3. compile both files (model and data) so as to generate an XCSP^3 instance (file)
4. solve the XCSP^3 file (problem instance under format XCSP^3) by using constraint solvers like, e.g., AbsCon, Choco, OscaR oar PicatSAT

This approach has many advantages:

- Python (and Java), JSON, 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 Python 3 (or Java 8) for modeling allows the user to avoid learning again a new programming language
- Using a coarse-grained XML structure permits to have compact and readable problem descriptions, easy to read for both humans and machines

Remark 1 *Using JSON instead of XML for representing instances is possible but has some drawbacks, as explained in an appendix of XCSP3 Specifications [4].*

Chapter 1

Illustrative Models in PyCSP^3

Warning. In this chapter, we introduce PyCSP^3 by means of various problems that illustrate the main ingredients of the library. We also usually show the result of compiling PyCSP^3 models into XCSP^3, although that part can be totally ignored.

1.1 Single Problems

We propose to start discovering PyCSP^3 with some very simple problems.

1.1.1 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 riddle 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 equations (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 (i.e., constraints involving exactly two distinct variables) and one quaternary constraint (i.e., constraint involving exactly four distinct variables).

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 because, by using that interval of values, we are absolutely

certain of not losing any solutions while avoiding to reason with infinite sets of values. We then obtain the following PyCSP^3 model in a file called 'Riddle.py':

PyCSP^3 Model 1

```
from pycsp3 import *

x1 = Var(range(15))
x2 = Var(range(15))
x3 = Var(range(15))
x4 = Var(range(15))

satisfy(
   x1 + 1 == x2,
   x2 + 1 == x3,
   x3 + 1 == x4,
   x1 + x2 + x3 + x4 == 14
)
```

In this Python file, after the first import statement, we declare stand-alone variables by using the PyCSP^3 function `Var()`. Here, we declare four variables called x1, x2, x3, and x4, each one with the set of integers $\{0, 1, \ldots, 14\}$ as domain.

Remark 2 *Currently in* PyCSP^3*, we can only define integer and symbolic variables, i.e., variables with a finite set of integers or symbols (strings).*

To define the domain of a variable, we can simply list values, or use `range()`. For example:

```
w = Var(range(15))
x = Var(0, 1)
y = Var(0, 2, 4, 6, 8)
z = Var("a", "b", "c")
```

declares four variables corresponding to:

- $w \in \{0, 1, \ldots, 14\}$
- $x \in \{0, 1\}$
- $y \in \{0, 2, 4, 6, 8\}$
- $z \in \{"a", "b", "c"\}$

Values can be directly listed as above, or given in a set as follows:

```
w = Var({range(15)})
x = Var({0, 1})
y = Var({0, 2, 4, 6, 8})
z = Var({"a", "b", "c"})
```

It is also possible to name the parameter `dom` when defining the domain:

```
w = Var(dom=range(15))
x = Var(dom={0, 1})
y = Var(dom={0, 2, 4, 6, 8})
z = Var(dom={"a", "b", "c"})
```

Finally, it is of course possible to use comprehension lists. For example, for y, we can write:

```
y = Var(i for i in range(10) if i % 2 == 0)
```

or equivalently:

```
y = Var({i for i in range(10) if i % 2 == 0})
```

or equivalently:

```
y = Var(dom={i for i in range(10) if i % 2 == 0})
```

Now, let us turn to constraints. When constraints must be imposed on variables, we say that these constraints must be satisfied. Then, to impose (post) them, we call the PyCSP^3 function `satisfy()`, with each constraint passed as a parameter (and so, with comma used as a separator between constraints). In our example, we have posted four constraints to be satisfied. These constraints are given in `intension`, by using classical arithmetic, relational and logical operators. Note that for forcing equality, we need to use '==' in Python (the operator '=' used for assignment cannot be redefined). In Tables 1.1 and 1.2, you can find the available operators and functions. In Table 1.3, you can find a few examples of `intension` constraints.

Once you have a PyCSP^3 model, you can compile it in order to get an XCSP^3 file that can be solved by a constraint solver. The command is as follows:

```
python3 Riddle.py
```

The content of the XCSP^3 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>
```

The variables in our model have been declared independently, but it is possible to declare them in a one-dimensional array. This gives a new PyCSP^3 model (variant) in a file called 'Riddle2.py':

PyCSP^3 Model 2

```
from pycsp3 import *

# x[i] is the ith integer of the sequence
x = VarArray(size=4, dom=range(15))

satisfy(
   x[0] + 1 == x[1],
   x[1] + 1 == x[2],
   x[2] + 1 == x[3],
   x[0] + x[1] + x[2] + x[3] == 14
)
```

and the XCSP^3 file obtained after compilation:

```
python3 Riddle2.py
```

is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i] is the ith integer of the sequence" size="[4]">
      0..14
    </array>
  </variables>
```

Arithmetic Operators	
+	addition
-	subtraction
*	multiplication
//	integer division
%	remainder
**	power

Relationnal Operators	
<	Less than
<=	Less than or equal
>=	Greater than or equal
>	Greater than
! =	Different from
==	Equal to

Set Operators	
`in`	membership
`not in`	non membership

Logical Operators	
~	logical not
|	logical or
&	logical and
^	logical xor

Table 1.1: Operators that can be used to build expressions (predicates) of `intension` constraints. Integer values 0 and 1 are respectively equivalent to Boolean values *false* and *true*.

Functions	
`abs`()	absolute value of the argument
`min`()	minimum value of 2 or more arguments
`max`()	maximum value of 2 or more arguments
`conjunction`()	conjunction of 2 or more arguments
`disjunction`()	disjunction of 2 or more arguments
`imply`()	implication between 2 arguments
`iff`()	equivalence between 2 or more arguments
`ift`()	ift(b,u,v) returns u if b is true, v otherwise

Table 1.2: Functions that can be used to build expressions (predicates) of `intension` constraints.

$x + y < 10$
$x * 2 - 10 * y + 5 == 100$
$\texttt{abs}(z[0] - z[1]) >= 2$
$(x == y) \,\vert\, (y == 0)$
$\texttt{disjunction}(x < 2, y < 4, x > y)$
$\texttt{imply}(x == 0, y > 0)$
$\texttt{iff}(x > 0, y > 0)$
$(x == 0) \,\hat{}\, (y == 1)$
$\texttt{ift}(x == 0, 5, 10)$

Table 1.3: A few examples of `intension` constraints.

```
  <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 name (id) is x, its size (length) is 4, and each of its variables has $\{0, 1, \ldots, 14\}$ as domain. Note that we use $x[i]$ for referring to the $(i + 1)$th variable of the array (since indexing starts at 0) and that any comment put in the line preceding the declaration of a variable (or variable array) is automatically inserted in the XCSP^3 file. The PyCSP^3 function for declaring an array of variables is `VarArray()` that requires two named parameters `size` and `dom`. For declaring a one-dimensional array of variables, the value of `size` must be an integer (or a list containing only one integer), for declaring a two-dimensional array of variables, the value of `size` must be a list containing exactly two integers, and so on.

In some situations, you may want to declare variables in an array with different domains. For a one-dimensional array, you can give the name of a function that accepts an integer i and returns the domain to be associated with the variable at index i in the array. For a two-dimensional array, you can give the name of a function that accepts a pair of integers (i, j) and returns the domain to be associated with the variable at indexes i, j in the array. And so on. For example, suppose that we have analytically deduced that the two first variables of the array x must be assigned a value strictly less than 6 and the two last variables of the array x must be assigned a value strictly less than 9. We can write:

PyCSP^3 Model 3

```
from pycsp3 import *

def domain_x(i):
    return range(6) if i < 2 else range(9)

# x[i] is the ith integer of the sequence
x = VarArray(size=4, dom=domain_x)

satisfy(
   x[0] + 1 == x[1],
   x[1] + 1 == x[2],
   x[2] + 1 == x[3],
   x[0] + x[1] + x[2] + x[3] == 14
)
```

The XCSP^3 file obtained after compilation is:

```
<instance format="XCSP3" type="CSP">
  <variables>
     <array id="x" note="x[i] is the ith integer of the sequence" 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>
```

Instead of calling named functions, we can use lambda functions. This gives:

PyCSP^3 Model 4

```
from pycsp3 import *

# x[i] is the ith integer of the sequence
x = VarArray(size=4, dom=lambda i: range(6) if i < 2 else range(9))

satisfy(
   x[0] + 1 == x[1],
   x[1] + 1 == x[2],
   x[2] + 1 == x[3],
   x[0] + x[1] + x[2] + x[3] == 14
)
```

Let us keep analyzing the code of our model. Because the three binary constraints are similar, one may wonder if we couldn't post these constraints together (in a list). This is indeed possible by using a comprehension list:

PyCSP^3 Model 5

```
from pycsp3 import *

# x[i] is the ith integer of the sequence
x = VarArray(size=4, dom=range(15))

satisfy(
   [x[i] + 1 == x[i + 1] for i in range(3)],

   x[0] + x[1] + x[2] + x[3] == 14
)
```

and the XCSP^3 file obtained after compilation is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i] is the ith integer of the sequence" 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>
```

Because of the presence of the comprehension list, we obtain a group of constraints in XCSP^3: basically, we have a constraint template with several parameters identified by %, and one "concrete" constraint per element `<args>` providing the effective arguments. For more information about groups in XCSP^3, see Chapter 10 in XCSP^3 Specifications. Of course, you can use the classical control structures of Python. So, an alternative way of writing the model is:

PyCSP^3 Model 6

```
from pycsp3 import *

# x[i] is the ith integer of the sequence
x = VarArray(size=4, dom=range(15))

for i in range(3):
   satisfy(
      x[i] + 1 == x[i + 1]
   )

satisfy(
   x[0] + x[1] + x[2] + x[3] == 14
)
```

Finally, it seems more appropriate to represent the last constraint as a constraint `sum`. We can then call the PyCSP^3 function `Sum()` that builds an object that can be compared for example with a value. This gives:

PyCSP^3 Model 7

```
from pycsp3 import *

# x[i] is the ith integer of the sequence
x = VarArray(size=4, dom=range(15))

satisfy(
   [x[i] + 1 == x[i + 1] for i in range(3)],

   Sum(x) == 14
)
```

and the XCSP^3 file obtained after compilation is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i] is the ith integer of the sequence" 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>
```

1.1.2 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 micro-structures). For example, Figure 1.1 presents the compatibility graph of a small constraint network P such that:

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

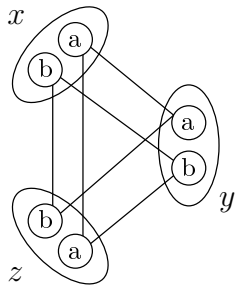


Figure 1.1: The compatibility graph of a small constraint network.

The interested reader can observe that the constraint network is arc-consistent (AC) but not path-inverse consistent (PIC). Don't worry! It doesn't matter here if you do not know anything about these properties. Anyway, the PyCSP^3 model for our problem, in a file called Pic.py, is as follows:

PyCSP^3 Model 8

```
from pycsp3 import *

x = Var("a", "b")
y = Var("a", "b")
z = Var("a", "b")


satisfy(
   (x, y) in {("a", "a"), ("b", "b")},
   (x, z) in {("a", "a"), ("b", "b")},
   (y, z) in {("a", "b"), ("b", "a")}
)
```

For compiling it, we execute:

```
python3 Pic.py
```

and the XCSP^3 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 in `extension`. In PyCSP^3, we simply use the operator `in` to represent such constraints: a tuple of variables representing the scope of the constraint is given at the left of the operator and a set of tuples of values is given at the right of the operator. This is basically what we write in mathematical form. Note that we use `in` when the constraint enumerates the allowed tuples (called supports) and `not in` when the constraint enumerates the forbidden tuples (called conflicts).

Now, suppose that instead of declaring symbolic variables, you prefer to declare integer variables. By replacing "a" by 0 and "b" by 1, you can write:

PyCSP^3 Model 9

```
from pycsp3 import *

x = Var(0, 1)
y = Var(0, 1)
z = Var(0, 1)

satisfy(
   (x, y) in {(0, 0), (1, 1)},
   (x, z) in {(0, 0), (1, 1)},
   (y, z) in {(0, 1), (1, 0)}
)
```

which, when compiled, gives:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <var id="x"> 0 1 </var>
    <var id="y"> 0 1 </var>
    <var id="z"> 0 1 </var>
  </variables>
  <constraints>
    <extension>
      <list> x y </list>
      <supports> (0,0)(1,1) </supports>
    </extension>
    <extension>
      <list> x z </list>
      <supports> (0,0)(1,1) </supports>
```

```
    </extension>
    <extension>
      <list> y z </list>
      <supports> (0,1)(1,0) </supports>
    </extension>
  </constraints>
</instance>
```

Note that the scope of an `extension` constraint is expected to be given under the form of a tuple, but can be given under the form of a list too. Similarly, the table of an `extension` constraint is expected to be given under the form of a set, but can be given under the form a list too. This means that, for example, it is possible to write:

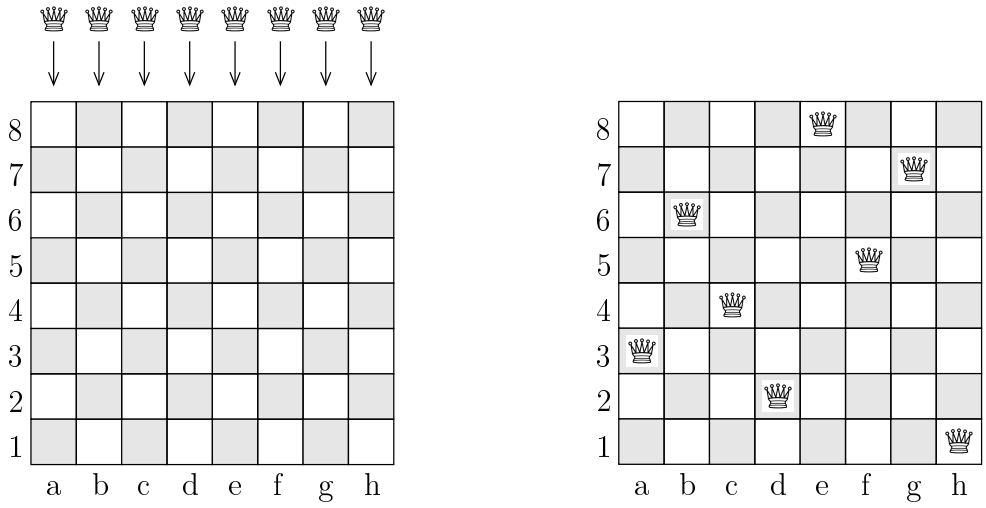
```
[x, y] in [(0, 0), (1, 1)]
```

1.2 Academic Problems

Contrary to single problems, academic problems require the introduction of some (elementary) pieces of data from the user.

1.2.1 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 1.2.



(a) Puzzle (b) Solution

Figure 1.2: 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 PyCSP^3, what you have to do is:

1. clearly identify the parameters of the problem (names and structures)
2. use these parameters in your model by using the fields of the predefined PyCSP^3 object `data`
3. specify effective values of these parameters when you compile to XCSP^3

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 can be translated into a PyCSP^3 model in a file 'Queens.py':

PyCSP^3 Model 10

```
from pycsp3 import *

n = data.n

# q[i] is the column of the ith queen (at row i)
q = VarArray(size=n, dom=range(n))

satisfy(
   (q[i] != q[j]) & (abs(q[i] - q[j]) != abs(i - j))
      for i in range(n) for j in range(i + 1, n)
)
```

Note how the parameter n is given by a field of the predefined PyCSP^3 object `data`; here, for simplicity, we prefer to copy its value in a local variable n. The `intension` constraints are given by a comprehension list (actually a generator, since brackets are omitted here although we could have inserted them).

Now, the question is: how can we solve a specific instance? The answer is: just compile the model while indicating with the option `-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 $n = 4$:

```
python3 Queens.py -data=4
```

and the XCSP^3 file obtained after compilation is:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="q" note="q[i] is the column of the ith queen (at row i)" size="[4]">
      0..3
    </array>
  </variables>
  <constraints>
    <group>
      <intension> and(ne(%0,%1),ne(abs(sub(%0,%1)),%2)) </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:

```
python3 Queens.py -data=queens-4.json
```

Remember that once you have an XCSP^3 file, you can run any solver that recognizes this format: AbsCon, Choco, PicatSAT, OscaR, ...

At this point, you have been told that it could be a good idea to post `allDifferent` constraints; remember that an `allDifferent` constraint imposes that all involved variables (or expressions) must take different values. It is known that it suffices to post three constraints as in the following model:

PyCSP^3 Model 11

```
from pycsp3 import *

n = data.n

# q[i] is the column of the ith queen (at row i)
q = VarArray(size=n, dom=range(n))

satisfy(
   # all queens are put on different columns
   AllDifferent(q),

   # no two queens on the same upward diagonal
   AllDifferent(q[i] + i for i in range(n)),

   # no two queens on the same downward diagonal
   AllDifferent(q[i] - i for i in range(n))
)
```

After compilation, we obtain:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="q" note="q[i] is the column of the ith queen (at row i)" size="[4]">
      0..3
    </array>
  </variables>
  <constraints>
    <allDifferent note="all queens are put on different columns">
      q[]
    </allDifferent>
    <allDifferent note="no two queens on the same upward diagonal">
      add(q[0],0) add(q[1],1) add(q[2],2) add(q[3],3)
    </allDifferent>
    <allDifferent note="no two queens on the same downward diagonal">
      sub(q[0],0) sub(q[1],1) sub(q[2],2) sub(q[3],3)
    </allDifferent>
  </constraints>
</instance>
```

Remark 3 *In* PyCSP^3*, most of the global constraints are posted by calling a function whose first letter is uppercase, as for example* `AllDifferent()`*,* `Sum()`*, and* `Cardinality()`*.*

Maybe, you think that it is annoying of having several files for various model variants (have you observed how many frameworks generate hundreds and even thousands of files; this is crazy!). In fact, you can put different model variants in the same file by using the PyCSP^3 function `variant()` that accepts a string as parameter (or nothing). When you compile, you can then indicate the name of the variant. Putting the two variants seen earlier in the same file 'Queens.py' gives:

PyCSP^3 Model 12

```
from pycsp3 import *

n = data.n

# q[i] is the column of the ith queen (at row i)
q = VarArray(size=n, dom=range(n))

if not variant():
   satisfy(
      # all queens are put on different columns
      AllDifferent(q),

      # no two queens on the same upward diagonal
      AllDifferent(q[i] + i for i in range(n)),

      # no two queens on the same downward diagonal
      AllDifferent(q[i] - i for i in range(n))
   )
elif variant("bin"):
    satisfy(
      (q[i] != q[j]) & (abs(q[i] - q[j]) != abs(i - j))
        for i in range(n) for j in range(i + 1, n)
    )
```

To compile the main model (variant), just type:

```
python3 Queens.py -data=4
```

To compile the model variant "bin", just type:

```
python3 Queens.py -variant=bin -data=4
```

1.2.2 Board Coloration

The (chess)board coloration problem is to color all squares of a board composed of r rows and c columns 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 1.3: Coloring Boards.

This time, we need two integer parameters `nRows` and `nCols`. For simplicity, their values (stored in two fields of the predefined PyCSP^3 object `data`) are copied in two local variables n and m. After a very rough analysis, we can decide to use $n \times m$ as an upper bound of the number of used colors. This gives a PyCSP^3 model in a file 'BoardColoration.py':

PyCSP^3 Model 13

```
from pycsp3 import *

n, m = data.nRows, data.nCols

# x[i][j] is the color at row i and column j
x = VarArray(size=[n, m], dom=range(n * m))

satisfy(
   NValues(x[i1][j1], x[i1][j2], x[i2][j1], x[i2][j2]) > 1
      for i1 in range(n) for i2 in range(i1 + 1, n)
      for j1 in range(m) for j2 in range(j1 + 1, m)
)

minimize(
   Maximum(x)
)
```

Here, we declare a two-dimensional array of variables: its name is x, its size is $n \times m$ and each of its variables has $\{0, 1, \ldots, n \times m - 1\}$ as domain. We need here to post several `notAllEqual` constraints. Actually, this constraint is a special case of the constraint `nValues`: we want that the number of different values taken by some variables (the scope of the constraint) is strictly greater than 1. This is given in the model by an expression involving the PyCSP^3 function `NValues`().

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. For an optimization problem, you can call either the PyCSP^3 function `minimize`() or the PyCSP^3 function `maximize`(). You can use different kinds of parameters:

- a stand-alone variable
- a general arithmetic expression, like in `u * 3 + v` where u and v are two variables
- a sum over a list (array) of variables by using the function `Sum`(), like in `Sum(x)`
- a dot product, like in `[u, v, w] * [2, 4, 3]` where u, v and w are three variables
- a minimum by using the function `Minimum`(), like in `Minimum(x)`
- a maximum by using the function `Maximum`(), like in `Maximum(x)`
- a number of different values by using the function `NValues`(), like in `NValues(x)`

To solve a specific instance, as usually, we have first to compile the model while indicating with the option `-data` either the values for `nRows` and `nCols` or the name of a JSON file containing an object with two fields `nRows` and `nCols`. In the former case, this gives for `nRows=3` and `nCols=4`:

```
python3 BoardColoration.py -data=[3,4]
```

Certainly, you wonder how values are associated with the fields of the predefined PyCSP^3 object `data`. Actually, the order of occurrences of these fields in the model (file) is used. The first occurrence

of a field of `data` in 'BoardColoration.py' is `data.nRows`, then it is `data.nCols`. So, we have automatically `data.nRows=3` and `data.nCols=4`.

However, you can relax this requirement by using names when specifying data, as for example, in:

```
python3 BoardColoration.py -data=[nCols=4,nRows=3]
```

The XCSP^3 file obtained after compilation is:

```
<instance format="XCSP3" type="COP">
  <variables>
    <array id="x" size="[3][4]" note="x[i][j] is the color at row i and col j">
      0..11
    </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[1][1] x[1][2] x[2][2] x[2][3] </args>
    </group>
  </constraints>
  <objectives>
    <minimize type="maximum"> x[][] </minimize>
  </objectives>
</instance>
```

We can also build a file "board-3-4.json" whose content is:

```
{
  "nRows": 3,
  "nCols": 4
}
```

and execute:

```
python3 BoardColoration.py -data=board-3-4.json
```

As a matter of fact, this problem has many symmetries. It is known that we can break variable symmetries by posting a lexicographic constraint between any two successive rows and any two successive columns. For posting lexicographic constraints, we can use the PyCSP^3 functions `LexIncreasing`() and `LexDecreasing`(). Besides, we can use two optional named parameters `strict` and `matrix` whose default values are `false`. When `matrix` is set to `True`, it means that the constraint must be applied on each row and each column of the specified two-dimensional array. On the other hand, it is relevant to tag this constraint because it clearly informs us that it is inserted for breaking symmetries: tagging is made possible by putting in a comment line an expression of the form `tag()`, with a token (or a sequence of tokens separated by a whitespace) between parentheses. The model is now:

PyCSP^3 Model 14

```
from pycsp3 import *

n, m = data.nRows, data.nCols

# x[i][j] is the color at row i and column j
x = VarArray(size=[n, m], dom=range(n * m))

satisfy(
   [NValues(x[i1][j1], x[i1][j2], x[i2][j1], x[i2][j2]) > 1
     for i1 in range(n) for i2 in range(i1 + 1, n)
     for j1 in range(m) for j2 in range(j1 + 1, m)],

   # tag(symmetry-breaking)
   LexIncreasing(x, matrix=True)
)

minimize(
   Maximum(x)
)
```

After compilation, we have the following additional element in the generated XCSP^3 file:

```
<lex class="symmetryBreaking">
  <matrix> x[][] </matrix>
  <operator> le </operator>
</lex>
```

Note the presence of the attribute `class` that results from the insertion of the expression `tag()`. Easily, a solver can now solve this instance with or without symmetry breaking. Indeed, at time of parsing, it is quite easy to discard XML elements with a specified tag (class): this is currently made possible with the available parsers in Java and C++ for XCSP^3. The interest is that we have only one file, with can be used for testing different model variations.

1.2.3 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 $i \in 0..n-1$ 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) by using an appropriate tag. This gives a PyCSP^3 model in a file 'MagicSequence.py':

PyCSP^3 Model 15

```
from pycsp3 import *

n = data.n

# x[i] is the ith value of the sequence
x = VarArray(size=n, dom=range(n))

satisfy(
   # each value i occurs exactly x[i] times in the sequence
   Cardinality(x, occurrences={i: x[i] for i in range(n)}),

   # tag(redundant-constraints)
   [
      Sum(x) == n,

      Sum((i - 1) * x[i] for i in range(n)) == 0
   ]
)
```

On the one hand, the constraint `cardinality` is exactly what we need here. Here, the PyCSP^3 function `Cardinality()` we use simply states that each value i in $0..n-1$ must occur exactly $x[i]$ times; a named parameter `occurrences` is set to a Python dictionary for storing that information. On the other hand, we have put together the two additional constraints in a list, permitting to tag these two constraints with the token "redundant-constraints".

Now, if we execute:

```
python3 MagicSequence.py -data=6
```

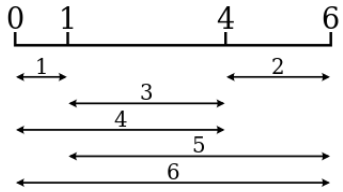
we obtain the following XCSP^3 instance:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i] is the ith value of the sequence" size="[6]">
      0..5
    </array>
  </variables>
  <constraints>
    <cardinality note="each value i occurs exactly x[i] times in the sequence">
      <list> x[] </list>
      <values> 0 1 2 3 4 5 </values>
      <occurs> x[] </occurs>
    </cardinality>
    <block class="redundant-constraints">
      <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>
```

1.2.4 Golomb Ruler

This problem (and its variants) is said to have many practical applications including sensor placements for x-ray crystallography and radio astronomy. A Golomb ruler is defined as a set of n integers

$0 = a_1 < a_2 < ... < a_n$ such that the $n \times (n-1)/2$ differences $a_j - a_i$, $1 \leq i < j \leq n$, are distinct. Such a ruler is said to contain n marks (or ticks) and to be of length a_n. The objective is to find optimal rulers (i.e., rulers of minimum length). An optimal ruler for $n = 4$ is illustrated below:



Dimitromanolakis has computed relatively short Golomb rulers and thus showed with computer aid that the optimal ruler for $n \leq 65,000$ has length less than n^2.

A simple model involves a single constraint `allDifferent`:

PyCSP^3 Model 16

```
from pycsp3 import *

n = data.n

# x[i] is the position of the ith tick
x = VarArray(size=n, dom=range(n * n))

satisfy(
   # all distances are different
   AllDifferent(abs(x[i] - x[j]) for i in range(n) for j in range(i + 1, n))
)

minimize(
   # minimizing the position of the rightmost tick
   Maximum(x)
)
```

One model variant involves auxiliary variables and ternary constraints. This variant shows how we can handle holes ("undefined" variables) in variable arrays. This variant is:

PyCSP^3 Model 17

```
from pycsp3 import *

n = data.n

# x[i] is the position of the ith tick
x = VarArray(size=n, dom=range(n * n))

def domain_y(i, j):
   return range(1, n * n) if i < j else None

# y[i][j] is the distance between x[i] and x[j] for i strictly less than j
y = VarArray(size=[n, n], dom=domain_y)

satisfy(
    # all distances are different
    AllDifferent(y),

    # linking variables from both arrays
    [x[j] == x[i] + y[i][j] for i in range(n) for j in range(i + 1, n)]
)
```

```
minimize(
    # minimizing the position of the rightmost tick
    Maximum(x)
)
```

Here, we declare a two-dimensional array of variables, called y, even if only the part in this array below the main diagonal really contains variables. This is handled by the auxiliary function `domain_y()` that returns an actual domain for a pair (i, j) when $i < j$, and `None` otherwise. This way, we can simply post a constraint `allDifferent` by specifying the array y (even if y contains some "undefined" cells/variables).

Of course, it is possible to use a lambda function when defining domains. Concerning symmetry breaking, we can decide to force $x[0]$ to be equal to 0, and to impose a strict increasing order on variables of x. When we want the values of a sequence of variables to be in increasing or decreasing order, we can call the PyCSP^3 functions `Increasing()` or `Decreasing()`; the named parameter `strict` can be used to indicate that the order must be strict. We obtain now:

PyCSP^3 Model 18

```
from pycsp3 import *

n = data.n

# x[i] is the position of the ith tick
x = VarArray(size=n, dom=range(n * n))

# y[i][j] is the distance between x[i] and x[j] for i strictly less than j
y = VarArray(size=[n, n], dom=lambda i, j: range(1, n * n) if i < j else None)

satisfy(
    # all distances are different
    AllDifferent(y),

    # linking variables from both arrays
    [x[j] == x[i] + y[i][j] for i in range(n) for j in range(i + 1, n)],

    # tag(symmetry-breaking)
    [x[0] == 0, Increasing(x, strict=True)]
)

minimize(
    # minimizing the position of the rightmost tick
    Maximum(x)
)
```

By executing:

```
python3 GolombRuler.py -data=4
```

we obtain:

```
<instance format="XCSP3" type="COP">
  <variables>
    <array id="x" note="x[i] is the position of the ith tick" size="[4]">
      0..16
    </array>
    <array id="y" note="y[i][j] is the distance between x[i] and x[j] for i strictly
        less than j" size="[4][4]">
      1..16
    </array>
```

```
  </variables>
  <constraints>
    <allDifferent note="all distances are different">
      y[0][1..3] y[1][2..3] y[2][3]
    </allDifferent>
    <group note="linking variables from both arrays">
      <intension> eq(%0,add(%1,%2)) </intension>
      <args> x[1] x[0] y[0][1] </args>
      <args> x[2] x[0] y[0][2] </args>
      <args> x[3] x[0] y[0][3] </args>
      <args> x[2] x[1] y[1][2] </args>
      <args> x[3] x[1] y[1][3] </args>
      <args> x[3] x[2] y[2][3] </args>
    </group>
    <block class="symmetry-breaking">
      <intension> eq(x[0],0) </intension>
      <ordered>
        <list> x[] </list>
        <operator> lt </operator>
      </ordered>
    </block>
  </constraints>
  <objectives>
    <minimize note="minimizing the position of the rightmost tick" type="maximum">
      x[]
    </minimize>
  </objectives>
</instance>
```

1.3 Structured Problems

Some problems need more than elementary data, that is more than a few elementary pieces of data such as integers. We call them structured problems.

1.3.1 Sudoku

	4							
5	3	9			1		6	
		1			2		5	
4		7	2		9			6
		6				5		
8			6		3	1		7
	8		7			2		
	6		3			4	1	8
							7	

(a) Puzzle

2	4	8	5	7	6	9	3	1
5	3	9	4	8	1	7	6	2
6	7	1	9	3	2	8	5	4
4	1	7	2	5	9	3	8	6
3	2	6	8	1	7	5	4	9
8	9	5	6	4	3	1	2	7
1	8	3	7	6	4	2	9	5
7	6	2	3	9	5	4	1	8
9	5	4	1	2	8	6	7	3

(b) Solution

Figure 1.4: Solving a Sudoku Grid

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 3)

An illustration is given by Figure 1.4.

Because there are several clues, and because their number cannot be anticipated, we need a parameter `clues` that represents a two-dimensional array of integer values. When `clues`$[i][j]$ 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). A PyCSP^3 model is given by the following file 'Sudoku.py':

PyCSP^3 Model 19

```
from pycsp3 import *

clues = data.clues  # if not 0, clues[i][j] is a value imposed at row i and col j

# x[i][j] is the value at row i and col j
x = VarArray(size=[9, 9], dom=range(1, 10))

satisfy(
   # imposing distinct values on each row and each column
   AllDifferent(x, matrix=True),

   # imposing distinct values on each block  tag(blocks)
   [AllDifferent(x[i:i + 3, j:j + 3]) for i in [0, 3, 6] for j in [0, 3, 6]]
)

if clues:
   satisfy(
      # imposing clues  tag(clues)
      x[i][j] == clues[i][j]
        for i in range(9) for j in range(9) if clues[i][j] > 0
    )
```

First, note how the named parameter `matrix` is used to ensure that all digits are different on each row and each column of the two-dimensional array x; this is the matrix version of `allDifferent`. Second, note how the notation $x[i : i + 3, j : j + 3]$ extracts a list of variables corresponding to a block of size 3×3 in x. This is similar to notations used in package NumPy and in library CPpy. Finally, each clue is naturally imposed under the form of a unary `intension` constraint.

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

```
{
  "clues": [
    [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 execute:

```
python3 Sudoku.py -data=grid.json
```

and we obtain the following XCSP^3 instance (simplified here):

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i][j] is the value at row i and col j" size="[9][9]">
      1..9
    </array>
  </variables>
  <constraints>
    <allDifferent note="imposing distinct values on each row and each column">
      <matrix> x[][] </matrix>
    </allDifferent>
    <group note="imposing distinct values on each block" class="blocks">
      <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 note="imposing clues" class="clues">
      <list> x[0][1] x[8][7] </list>   // only two of them inserted here for conciseness
      <values> 4 7 </values>
    </instantiation>
  </constraints>
</instance>
```

Once again, we have used tags. This way, it will be easy at parsing time to discard blocks or clues, if wished. Suppose now that we want to generate an instance without any clue. Of course, we can build a grid only containing the value 0, but this is a little bit tedious. Actually, you just need to use a JSON file like this:

```
{
  "clues": null
}
```

An alternative is simply to execute:

```
python3 Sudoku.py -data=None
```

1.3.2 Warehouse Location

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 for more information.

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]
```

Figure 1.5: Warehouse.

```
    ]
}
```

A PyCSP^3 model of this problem is given by the following file 'Warehouse.py':

PyCSP^3 Model 20

```
from pycsp3 import *

cost = data.fixedCost  # for each open warehouse
capacities = data.warehouseCapacities
costs = data.storeSupplyCosts
nWarehouses, nStores = len(capacities), len(costs)

# w[i] is the warehouse supplying the ith store
w = VarArray(size=nStores, dom=range(nWarehouses))

# c[i] is the cost of supplying the ith store
c = VarArray(size=nStores, dom=lambda i: set(costs[i]))

# o[j] is 1 if the jth warehouse is open
o = VarArray(size=nWarehouses, dom={0, 1})

satisfy(
    # capacities of warehouses must not be exceeded
    [Count(w, value=j) <= capacities[j] for j in range(nWarehouses)],

    # the warehouse supplier of the ith store must be open
    [o[w[i]] == 1 for i in range(nStores)],

    # computing the cost of supplying the ith store
    [costs[i][w[i]] == c[i] for i in range(nStores)]
)

minimize(
    # minimizing the overall cost
    Sum(c) + Sum(o) * cost
)
```

Here, we associate a specific domain with each variable of the array c by means of a lambda function. For dealing with warehouse capacities, we use the constraint `count`: the number of variables in a given list (here, w) that take the value specified by the named parameter `value` must be less than a constant. For linking stores with warehouses, we use the constraint `element`: the variable in the array o at index $w[i]$ must be 1 because this variable denotes the warehouse supplying the

ith store, and it must be open. Note that the index is not a constant but a variable of our model. Similarly, we use the constraint `element` for computing the actual costs; this time the array contains values (and not variables) and the target to reach is given by a variable. Finally, the objective function corresponds to minimizing two partial sums.

After executing:

```
python3 Warehouse.py -data=warehouse.json
```

we obtain the following XCSP^3 instance (some parts are omitted; see the presence of ellipsis):

```
<instance format="XCSP3" type="COP">
  <variables>
    <array id="w" note="w[i] is the warehouse supplying the ith store" size="[10]">
      0..4
    </array>
    <array id="c" note="c[i] is the cost of supplying the ith store" size="[10]">
      <domain for="c[0]"> 11 24 25 30 100 </domain>
      <domain for="c[1]"> 27 28 74 82 83 </domain>
      ...  // ellipsis
    </array>
    <array id="o" note="o[j] is 1 if the jth warehouse is open" size="[5]">
      0 1
    </array>
  </variables>
  <constraints>
    <block note="capacities of warehouses must not be exceeded">
      <count>
        <list> w[] </list>
        <values> 0 </values>
        <condition> (le,1) </condition>
      </count>
      ... // ellipsis
    </block>
    <group note="the warehouse supplier of the ith store must be open">
      <element>
        <list> o[] </list>
        <index> %0 </index>
        <value> 1 </value>
      </element>
      <args> w[0] </args>
      <args> w[1] </args>
      ...  // ellipsis
    </group>
    <block note="computing the cost of supplying the ith store">
      <element>
        <list> 100 24 11 25 30 </list>
        <index> w[0] </index>
        <value> c[0] </value>
      </element>
      ... // ellipsis
    </block>
  </constraints>
  <objectives>
    <minimize note="minimizing the overall cost" 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>
```

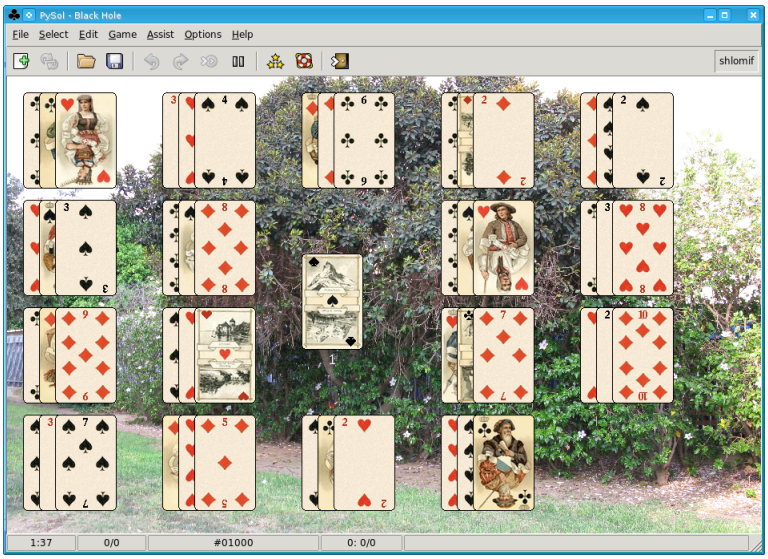


Figure 1.6: Blackhole.

1.3.3 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 1.6.

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

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

A PyCSP^3 model of this problem is given by the following file 'Blackhole.py':

PyCSP^3 Model 21

```
from pycsp3 import *

m, piles = data.nCardsPerSuit, data.piles
nCards = 4 * m

# x[i] is the value j of the card at position i of the stack
x = VarArray(size=nCards, dom=range(nCards))

# y[j] is the position i of the card whose value is j
y = VarArray(size=nCards, dom=range(nCards))

table = {(i, j) for i in range(nCards) for j in range(nCards)
```

```
                if i % m == (j + 1) % m or j % m == (i + 1) % m}

  satisfy(
     Channel(x, y),

     # the Ace of Spades is initially put on the stack
     y[0] == 0,

     # cards must be played in the order of the piles
     [Increasing([y[j] for j in pile], strict=True) for pile in piles],

     # each new card put on the stack must be at a higher or lower rank
     [(x[i], x[i + 1]) in table for i in range(nCards - 1)]
  )
```

Note how the `channel` constraint is used to make a channeling between the two arrays x and y, how the value of the first variable of y is imposed by a unary `intension` constraint, how we guarantee to take cards from each pile in a strict increasing order with `increasing` constraints and how `extension` constraints are posted after having precomputed a table.

Because the same table constraint is imposed on successive pairs of variables, we can use the meta-constraint `slide`, which may be useful to solvers. It suffices to replace the last argument of `satisfy()` with:

```
Slide((x[i], x[i + 1]) in table for i in range(nCards - 1))
```

With this meta-constraint `slide`, after executing:

```
python3 Blackhole.py -data=blackhole.json
```

we obtain the following XCSP^3 instance:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i] is the value j of the card at position i 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 of Spades is initially put on the stack"> eq(y[0],0) </
        intension>
    <group note="cards must be played in the order of the piles">
      <ordered>
        <list> %0 %1 %2 </list>
        <operator> lt </operator>
      </ordered>
      <args> y[1] y[4] y[13] </args>
      <args> y[15] y[9] y[6] </args>
      <args> y[14] y[2] y[12] </args>
      <args> y[7..8] y[5] </args>
      <args> y[11] y[10] y[3] </args>
    </group>
    <slide note="each new card put on the stack must be at a higher or lower rank">
      <list> x[] </list>
```

```
    <extension>
      <list> %0 %1 </list>
      <supports> (0,1)(0,3)(0,5)(0,7)(0,9)(0,11)(0,13)(0,15)(1,0)(1,2)(1,4)(1,6)
          (1,8)(1,10)(1,12)(1,14)(2,1)(2,3)(2,5)(2,7)(2,9)(2,11)(2,13)(2,15)(3,0)
          (3,2)(3,4)(3,6)(3,8)(3,10)(3,12)(3,14)(4,1)(4,3)(4,5)(4,7)(4,9)(4,11)
          (4,13)(4,15)(5,0)(5,2)(5,4)(5,6)(5,8)(5,10)(5,12)(5,14)(6,1)(6,3)(6,5)
          (6,7)(6,9)(6,11)(6,13)(6,15)(7,0)(7,2)(7,4)(7,6)(7,8)(7,10)(7,12)(7,14)
          (8,1)(8,3)(8,5)(8,7)(8,9)(8,11)(8,13)(8,15)(9,0)(9,2)(9,4)(9,6)(9,8)
          (9,10)(9,12)(9,14)(10,1)(10,3)(10,5)(10,7)(10,9)(10,11)(10,13)(10,15)
          (11,0)(11,2)(11,4)(11,6)(11,8)(11,10)(11,12)(11,14)(12,1)(12,3)(12,5)
          (12,7)(12,9)(12,11)(12,13)(12,15)(13,0)(13,2)(13,4)(13,6)(13,8)(13,10)
          (13,12)(13,14)(14,1)(14,3)(14,5)(14,7)(14,9)(14,11)(14,13)(14,15)(15,0)
          (15,2)(15,4)(15,6)(15,8)(15,10)(15,12)(15,14) </supports>
    </extension>
  </slide>
</constraints>
</instance>
```

1.3.4 Rack Configuration

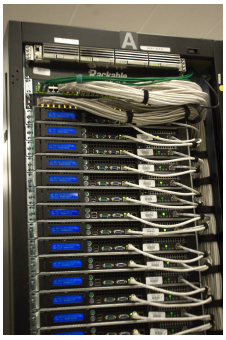


Figure 1.7: Rack.

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 characterized by the power it requires. Each rack model is characterized by the maximal power it can supply, its size (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 minimized.

See CSPLib–Problem 031 for more information.

An example of data is given by the file 'rack.json' containing:

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

A PyCSP^3 model for this problem is given by the following file 'Rack.py':

PyCSP^3 Model 22

```
from pycsp3 import *

nRacks, models, cardTypes = data.nRacks, data.models, data.cardTypes

# we add first a dummy model (0,0,0)
models = [(0, 0, 0)] + [tuple(model) for model in models]
nModels, nTypes = len(models), len(cardTypes)

powers = [row[0] for row in models]
sizes = [row[1] for row in models]
costs = [row[2] for row in models]
cardPowers = [row[0] for row in cardTypes]
cardDemands = [row[1] for row in cardTypes]

# m[i] is the model used for the ith rack
m = VarArray(size=nRacks, dom=range(nModels))

# nc[i][j] is the number of cards of type j put in the ith rack
nc = VarArray(size=[nRacks, nTypes],
              dom=lambda i, j: range(min(max(sizes), cardDemands[j]) + 1))

# p[i] is the power of the ith rack
p = VarArray(size=nRacks, dom=set(powers))

# s[i] is the size of the ith rack
s = VarArray(size=nRacks, dom=set(sizes))

# c[i] is the cost of the ith rack
c = VarArray(size=nRacks, dom=set(costs))

satisfy(
   # linking model and power of the ith rack
   [(m[i], p[i]) in enumerate(powers) for i in range(nRacks)],

   # linking model and size of the ith rack
   [(m[i], s[i]) in enumerate(sizes) for i in range(nRacks)],

   # linking model and cost of the ith rack
   [(m[i], c[i]) in enumerate(costs) for i in range(nRacks)],

   # connector-capacity constraints
   [Sum(nc[i]) <= s[i] for i in range(nRacks)],

   # power-capacity constraints
   [nc[i] * cardPowers <= p[i] for i in range(nRacks)],

   # demand constraints
   [Sum(nc[:, j]) == cardDemands[j] for j in range(nTypes)],

   # tag(symmetry-breaking)
   [
      Decreasing(m),
      (m[0] != m[1]) | (nc[0][0] >= nc[1][0])
   ]
)

minimize(
   # minimizing the total cost paid for all racks
   Sum(c)
)
```

After initially building some useful lists, five arrays of variables are declared. Then, three lists of `extension` constraints are posted. Note how the Python function `enumerate`() allows us to build tables that are relevant for our model. Then, three lists of `sum` constraints are posted. In the second list, we use a dot product, and in the third list, we use the notation $nc[:, j]$ to extract the jth column of the array nc, as in NumPy.

Concerning the function `enumerate`() called in the context of an `extension` constraint, the returned object returned is automatically transformed into a set.

```
>>> t = [2, 3, 4]
>>> enumerate(t)
<enumerate object at 0x7f42c5148708>
>>> set(enumerate(t))
{(0, 2), (1, 3), (2, 4)}
```

As usual, for generating an XCSP^3 instance, we just need to execute:

```
python3 Rack.py -data=rack.json
```

One drawback with the previous model is that it is difficult to understand the role of each piece of data, when looking independently at the JSON file. One remedy is then to choose a clearer structure as in this file 'rack2.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}
  ]
}
```

In PyCSP^3, it is quite easy to change the representation (structure) of data. It suffices to update the way the predefined PyCSP^3 object `data` is used in the model. In our case, with this new representation, we modify the first part of our file 'Rack.py' as follows:

PyCSP^3 Model 23

```
from pycsp3 import *

nRacks, models, cardTypes = data.nRacks, data.rackModels, data.cardTypes

# we add first a dummy model (0,0,0)
models = [{'power': 0, 'nConnectors': 0, 'price': 0}] + models
nModels, nTypes = len(models), len(cardTypes)

powers = [model['power'] for model in models]
sizes = [model['nConnectors'] for model in models]
costs = [model['price'] for model in models]
cardPowers = [cardType['power'] for cardType in cardTypes]
cardDemands = [cardType['demand'] for cardType in cardTypes]

...  # the rest of the model remains the same
```

Chapter 2

Data, Variables and Objectives

This chapter is currently missing. But note that many examples have already been given all along other chapters.

Chapter 3

Twenty Popular Constraints

In this chapter, we introduce some popular constraints, those recognized by many constraint solvers (and from XCSP^3-core). Figure 3.1 shows their classification. Some sections of this chapter are currently missing (but are planned to be totally written by mid-January 2010).

Semantics. Concerning the semantics, here are a few important remarks:

- when presenting the semantics, we distinguish between a variable x and its assigned value $\boldsymbol{x}$ (note the bold face on the symbol x).
- in many constraints, quite often, we need to introduce numerical conditions (comparisons) composed of an operator $\odot$ in $\{<, \leq, >, \geq, =, \neq, \in, \notin\}$ and a right-hand side operand k that can be a value (constant), a variable of the model, an interval or a set; the left-hand side being indirectly defined by the constraint. The numerical condition is a kind of terminal operation to be applied after the constraint has "performed some computation". In Python, the operator $\odot$ is from $\{<, <=, >, >=, ==, !=, \texttt{in}, \texttt{not in}\}$ and an interval is given by a `range` object. For example, constraints involving numerical conditions are $\texttt{Sum(x)} > 10$, $\texttt{Count(x, value} = 1) \texttt{ in range(10)}$, $\texttt{NValues(x) in } \{2, 4, 6\}$ and $\texttt{Minimum(x)} == \texttt{y}$. Of course, we can also write $10 < \texttt{Sum(x)}$ and $\texttt{y} == \texttt{Minimum(x)}$, but for simplicity of the presentation, we shall always assume that numerical conditions are on the right side. For the semantics of a numerical condition $(\odot, k)$, and depending on the form of k (a value, a variable, an interval or a set), we shall indiscriminately use $\boldsymbol{k}$ to denote the value of the constant k, the value of the variable k, the interval $l..u$ represented by k, or the set $\{a_1, \ldots, a_p\}$ represented by k.

3.1 Constraint `intension`

An `intension` constraint corresponds to a Boolean expression, which is usually called predicate. For example, the constraint $x + y = z$ corresponds to an equation, which is an expression evaluated to *false* or *true* according to the values assigned to the variables x, y and z. However, note that for equality, we need to use '==' in Python (the operator '=' used for assignment cannot be redefined), and so, the previous constraint must be written $x + y == z$ in PyCSP^3. To build predicates, classical arithmetic, relational and logical operators (and functions) are available; they are presented in Table1.1 and Table 1.2. In Table 1.3, you can find a few examples of `intension` constraints. Note that the integer values 0 and 1 are respectively equivalent to the Boolean values *false* and *true* This allows us to combine Boolean expressions with arithmetic operators (for example, addition) without requiring any type conversions. For example, it is valid to write $(x < 5) + (y < z) == 1$ for stating that exactly one of the Boolean expressions $x < 5$ and $y < z$ must be true, although it may be possible (and more relevant) to write it differently.

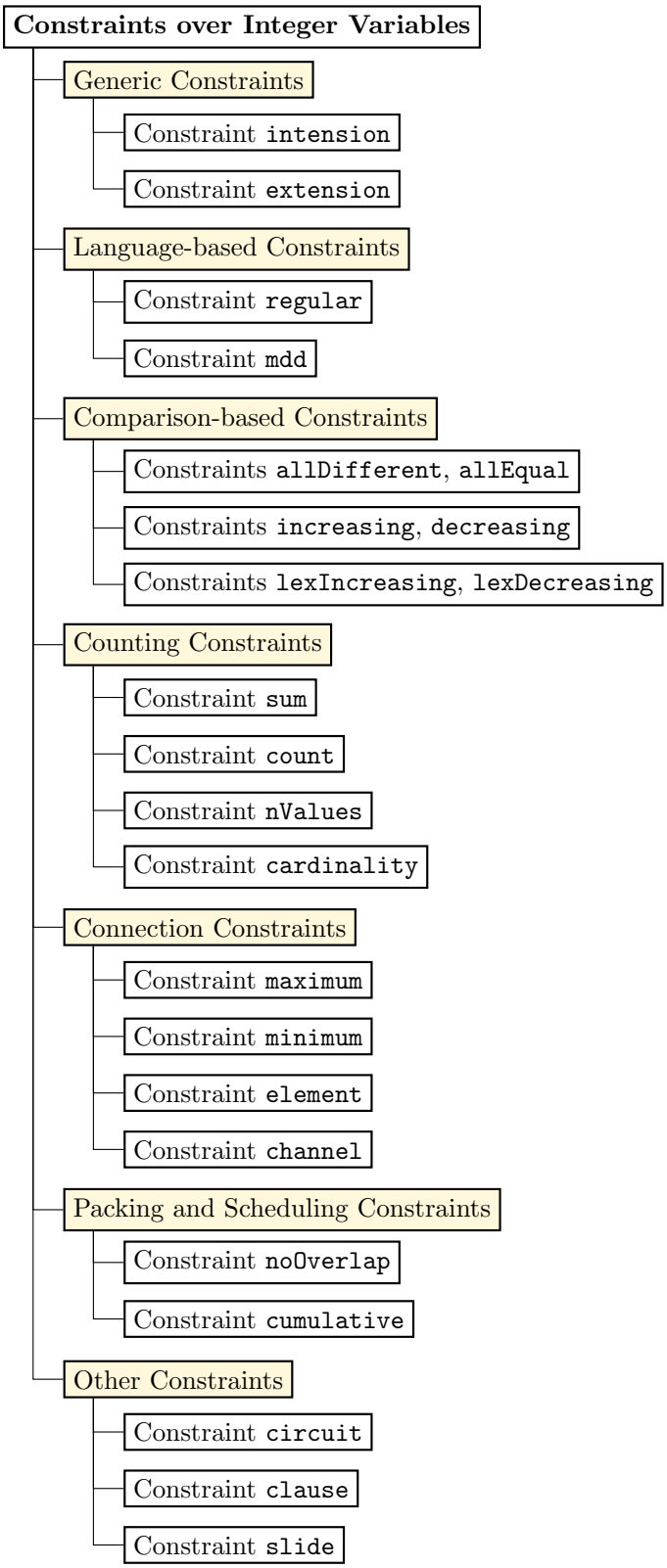


Figure 3.1: Popular constraints over integer variables.

Below, P denotes a predicate expression with r formal parameters (not shown here, for simplicity), $X = \langle x_0, x_1, \ldots, x_{r-1} \rangle$ denotes a sequence of r variables, the scope of the constraint, and $P(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{r-1})$ denotes the value (0/false or 1/true) returned by P for a specific instantiation of the variables of X.

Semantics 1

`intension`(X, P)`, with` $X = \langle x_0, x_1, \ldots, x_{r-1} \rangle$ `and` P `a predicate iff`
$P(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{r-1}) = true$ (1) `// recall that 1 is equivalent to true`

Zebra. The Zebra puzzle (sometimes referred to as Einstein's puzzle) is defined as follows. There are five houses in a row, numbered from left to right. Each of the five houses is painted a different color, and has one inhabitant. The inhabitants are all of different nationalities, own different pets, drink different beverages and have different jobs.

Figure 3.2: In which house lives the zebra?

We know that:

- colors are yellow, green, red, white, and blue
- nations of inhabitants are italy, spain, japan, england, and norway
- pets are cat, zebra, bear, snails, and horse
- drinks are milk, water, tea, coffee, and juice
- jobs are painter, sculptor, diplomat, pianist, and doctor
- The painter owns the horse
- The diplomat drinks coffee
- The one who drinks milk lives in the white house
- The Spaniard is a painter
- The Englishman lives in the red house
- The snails are owned by the sculptor
- The green house is on the left of the red one
- The Norwegian lives on the right of the blue house
- The doctor drinks milk

- The diplomat is Japanese
- The Norwegian owns the zebra
- The green house is next to the white one
- The horse is owned by the neighbor of the diplomat
- The Italian either lives in the red, white or green house

A PyCSP^3 model of this problem is given by the following file 'Zebra.py':

PyCSP^3 Model 24

```
from pycsp3 import *

# colors[i] is the color of the ith house
yellow, green, red, white, blue = colors = VarArray(size=5, dom=range(5))

# nations[i] corresponds to the nationality of the inhabitant of the ith house
italy, spain, japan, england, norway = nations = VarArray(size=5, dom=range(5))

# jobs[i] is the job of the inhabitant of the ith house
painter, sculptor, diplomat, pianist, doctor = jobs = VarArray(size=5, dom=range(5))

# pets[i] is the pet of the inhabitant of the ith house
cat, zebra, bear, snails, horse = pets = VarArray(size=5, dom=range(5))

# drinks[i] is the drink of the inhabitant of the ith house
milk, water, tea, coffee, juice = drinks = VarArray(size=5, dom=range(5))

satisfy(
   AllDifferent(colors),
   AllDifferent(nations),
   AllDifferent(jobs),
   AllDifferent(pets),
   AllDifferent(drinks),

   painter == horse,
   diplomat == coffee,
   white == milk,
   spain == painter,
   england == red,
   snails == sculptor,
   green + 1 == red,
   blue + 1 == norway,
   doctor == milk,
   japan == diplomat,
   norway == zebra,
   abs(green - white) == 1,
   horse in {diplomat - 1, diplomat + 1},
   italy in {red, white, green}
)
```

In this model, there are many equations. Note also the possibility of using the operator `in`.

3.2 Constraint `extension`

An `extension` constraint is often referred to as a `table` constraint. It is defined by enumerating (in a set) the tuples of values that are allowed (tuples are called supports) or forbidden (tuples are called conflicts) for a sequence of variables.

A (positive) table constraint is then defined by a scope (a sequence or tuple of variables) ⟨`scope`⟩ and a table (set of tuples of values) ⟨`table`⟩ as follows:

⟨`scope`⟩ ∈ ⟨`table`⟩

When the table constraint is negative (i.e., enumerates forbidden tuples), we have:

⟨`scope`⟩ ∉ ⟨`table`⟩

With X denoting a scope (sequence or tuple of variables), and S and C denoting sets of supports and conflicts, we have the following semantics for non-unary *positive* table constraints:

Semantics 2

`extension`(X, S)`, with` $X = \langle x_0, x_1, \ldots, x_{r-1} \rangle$ `and` S `a set of supports, iff`
$\langle \boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{r-1} \rangle \in S$

Prerequisite : $\forall \tau \in S, |\tau| = |X| \geq 2$

and this one for non-unary *negative* table constraints:

Semantics 3

`extension`(X, C)`, with` $X = \langle x_0, x_1, \ldots, x_{r-1} \rangle$ `and` C `a set of conflicts, iff`
$\langle \boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{r-1} \rangle \notin C$

Prerequisite : $\forall \tau \in C, |\tau| = |X| \geq 2$

In PyCSP^3, we can directly write table constraints in mathematical forms, by using tuples, sets and the operators `in` and `not in`. The scope is given by a tuple of variables on the left of the constraining expression and the table is given by a set of tuples of values on the right of the constraining expression. Although not recommended, it is possible to write scopes and tables under the form of lists.

Traffic Lights. From CSPLib: "Consider a four way traffic junction with eight traffic lights. Four of the traffic lights are for the vehicles and can be represented by the variables $v1$ to $v4$ with domains $\{r, ry, g, y\}$ (for red, red-yellow, green and yellow). The other four traffic lights are for the pedestrians and can be represented by the variables $p1$ to $p4$ with domains $\{r, g\}$. The constraints on these variables can be modelled by quaternary constraints on (v_i, p_i, v_j, p_j) for $1 \leq i \leq 4, j = (1+i) \bmod 4$ which allow just the tuples $\{(r, r, g, g), (ry, r, y, r), (g, g, r, r), (y, r, ry, r)\}$."

PyCSP^3 Model 25

```
from pycsp3 import *

R, RY, G, Y = "red", "red-yellow", "green", "yellow"

# v[i] is the color for the ith vehicle traffic light
v = VarArray(size=4, dom={R, RY, G, Y})

# p[i] is the color for the ith pedestrian traffic light
p = VarArray(size=4, dom={R, G})

table = {(R, R, G, G), (RY, R, Y, R), (G, G, R, R), (Y, R, RY, R)}
```

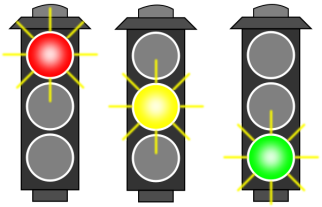


Figure 3.3: How to adjust traffic lights? (image from freesvg.org)

```
satisfy(
   (v[i], p[i], v[(i + 1) % 4], p[(i + 1) % 4]) in table for i in range(4)
)
```

Note how we naturally build a set of tuples (with symbolic values, here). Four quaternary table constraints are posted in this model.

Traveling Tournament with Predefined Venues. From CSPLib: "The TTPPV was introduced in [13] and consists of finding an optimal compact single round robin schedule for a sport event. Given a set of n teams, each team has to play against every other team exactly once. In each round, a team plays either at home or away, however no team can play more than two (or three) consecutive times at home or away. The sum of the traveling distance of each team has to be minimized. The particularity of this problem resides on the venue of each game that is predefined, i.e. if team a plays against b it is already known whether the game is going to be held at a's home or at b's home. The original instances assume symmetric circular distances: for $i \leq j, d_{i,j} = d_{j,i} = \min(j-i, i-j+n)$."

An example of data is given by the following JSON file:

```
{
  "nTeams": 8,
  "predefinedVenues": [
    [0,1,1,0,0,0,0,1],
    [0,0,0,1,0,1,0,1],
    ...
  ]
}
```

A PyCSP^3 model of this problem is given by the following file:

PyCSP^3 Model 26

```
from pycsp3 import *

pv = data.predefinedVenues
nTeams, nRounds = data.nTeams, data.nTeams - 1

def cdist(i, j):  # circular distance between i and j
   return min(abs(i - j), nTeams - abs(i - j))

def table_end(i):
```

```
    # when playing at home (whatever the opponent, travel distance is 0)
    return {(1, ANY, 0)} | {(0, j, cdist(i, j)) for j in range(nTeams) if j != i}

def table_intern(i):
    return ({(1, 1, ANY, ANY, 0)} |
       {(0, 1, j, ANY, circ_distance(j, i)) for j in range(nTeams) if j != i} |
       {(1, 0, ANY, j, circ_distance(i, j)) for j in range(nTeams) if j != i} |
       {(0, 0, j1, j2, circ_distance(j1, j2)) for j1 in range(nTeams)
         for j2 in range(nTeams) if different_values(i, j1, j2)})

def automaton():
    tr = {("q", 0, "q01"), ("q", 1, "q11"), ("q01", 0, "q02"), ("q01", 1, "q11"),
      ("q11", 0, "q01"), ("q11", 1, "q12"), ("q02", 1, "q11"), ("q12", 0, "q01")}
    return Automaton(start="q", final={"q01", "q02", "q11", "q12"}, transitions=tr)


# o[i][k] is the opponent (team) of the ith team  at the kth round
o = VarArray(size=[nTeams, nRounds], dom=range(nTeams))

# h[i][k] is 1 iff the ith team plays at home at the kth round
h = VarArray(size=[nTeams, nRounds], dom={0, 1})

# t[i][k] is the travelled distance by the ith team at the kth round.
# An additional round is considered for returning at home.
t = VarArray(size=[nTeams, nRounds + 1], dom=range(nTeams // 2 + 1))

satisfy(
    # a team cannot play against itself
    [o[i][k] != i for i in range(nTeams) for k in range(nRounds)],

    # ensuring predefined venues
    [pv[i][o[i][k]] == h[i][k] for i in range(nTeams) for k in range(nRounds)],

    # ensuring symmetry of games: if team i plays against j, then j plays against i
    [o[:, k][o[i][k]] == i for i in range(nTeams) for k in range(nRounds)],

    # each team plays once against all other teams
    [AllDifferent(row) for row in o],

    # at most 2 consecutive games at home, or consecutive games away
    [h[i] in automaton() for i in range(nTeams)],

    # handling travelling for the first game
    [(h[i][0], o[i][0], t[i][0]) in table_end(i) for i in range(nTeams)],

    # handling travelling for the last game
    [(h[i][- 1], o[i][- 1], t[i][-1]) in table_end(i) for i in range(nTeams)],

    # handling travelling for two successive games
    [(h[i][k], h[i][k + 1], o[i][k], o[i][k + 1], t[i][k + 1]) in table_intern(i)
      for i in range(nTeams) for k in range(nRounds - 1)]
)

minimize(
    # minimizing summed up travelled distance
    Sum(t)
)
```

Two functions, called `table_end`() and `table_intern`(), are introduced here to build short tables, i.e., tables that contain short tuples (a short tuple is a tuple containing the special symbol '*', denoted in PyCSP^3 by the constant ANY). When the symbol '*' is present, it means that any value from the domain of the corresponding variable can be present at its position. For more information about

short tables, see e.g., [11, 16]. Remember that the symbol | is used by Python to perform the union of two sets, and that we use the notation $o[:, k]$ to extract the kth column of the array o, as in NumPy. Some `regular` constraints (based on automata) are also posted, but we shall discuss them in the next section.

Subgraph Isomorphism. An instance of the *subgraph isomorphism problem* is defined by a pattern graph $G_p = (V_p, E_p)$ and a target graph $G_t = (V_t, E_t)$: the objective is to determine whether G_p is isomorphic to some subgraph(s) in G_t. Finding a solution to such a problem instance means then finding a *subisomorphism function*, that is an injective mapping $f : V_p \to V_t$ such that all edges of G_p are preserved: $\forall (v, v') \in E_p, (f(v_p), f(v'_p)) \in E_t$. Here, we refer to the partial, and not the induced subgraph isomorphism problem. An example of data is given by the following JSON file:

```
{
  "nPatternNodes": 180,
  "nTargetNodes": 200,
  "patternEdges":[[0,1], [0,3], [0,17], ... ],
  "targetEdges":[[0,34], [0,65], [0,129], ...]
}
```

A PyCSP^3 model of this problem is given by the following file:

PyCSP^3 Model 27

```
from pycsp3 import *

n, m = data.nPatternNodes, data.nTargetNodes

def structures():
    pe, te = data.patternEdges, data.targetEdges
    p_degrees = [len([e for e in pe if i in e]) for i in range(n)]
    t_degrees = [len([e for e in te if i in e]) for i in range(m)]
    tab = {(i, j) for (i, j) in te} | {(j, i) for (i, j) in te}
    cfs = [[j for j in range(m) if t_degrees[j] < p_degrees[i]] for i in range(n)]
    pl, tl = [i for (i, j) in pe if i == j], [i for (i, j) in te if i == j]
    return pl, tl, tab, cfs

p_loops, t_loops, table, degree_conflicts = structures()

# x[i] is the target node to which the ith pattern node is mapped
x = VarArray(size=n, dom=range(m))

satisfy(
    # ensuring injectivity
    AllDifferent(x),

    # preserving edges
    [(x[i], x[j]) in table for (i, j) in data.patternEdges],

    # being careful of self-loops
    [x[i] in t_loops for i in p_loops],

    # tag(redundant-constraints)
    [x[i] not in tab for i, tab in enumerate(degree_conflicts) if len(tab) > 0]
)
```

In this model, some binary `extension` constraints are posted for preserving edges, and some unary `extension` constraints are posted for handling self-loops as well as for reducing domains by reasoning from node degrees. Note that for a unary `extension` constraint, we use the form: x `in` S where x is a variable of the model and S a set of values.

3.3 Constraint `regular`

Definition 1 (DFA) *A* deterministic finite automaton *(DFA) is defined by a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$ *where* Q *is a finite set of states,* Σ *is a finite set of symbols called the alphabet,* $\delta : Q \times \Sigma \to Q$ *is a transition function,* $q_0 \in Q$ *is the initial state, and* $F \subseteq Q$ *is the set of final states.*

Given an input string (a finite sequence of symbols taken from the alphabet Σ), the automaton starts in the initial state q_0, and for each symbol in sequence of the string, applies the transition function to update the current state. If the last state reached is a final state then the input string is accepted by the automaton. The set of strings that the automaton A accepts constitutes a language, denoted by $L(A)$, which is technically a regular language. When the automaton is non-deterministic, we can find two transitions (q_i, a, q_j) and (q_i, a, q_k) such that $q_j \neq q_k$.

A `regular` constraint [6, 14] ensures that the sequence of values assigned to the variables of its scope must belong to a given regular language (i.e., forms a word that can be recognized by a deterministic, or non-deterministic, finite automaton). For such constraints, a DFA is then used to determine whether or not a given tuple is accepted. This can be an attractive approach when constraint relations can be naturally represented by regular expressions in a known regular language. For example, in rostering problems, regular expressions can represent valid patterns of activities. The semantics is:

Semantics 4

`regular`(X, A)`, with` $X = \langle x_0, x_1, \ldots, x_{r-1} \rangle$ `and` A `a finite automaton, iff`
$\boldsymbol{x}_0 \boldsymbol{x}_1 \ldots \boldsymbol{x}_{r-1} \in L(A)$

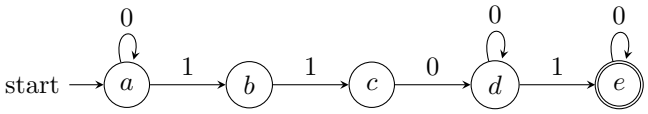
In PyCSP^3, we can directly write `regular` constraints in mathematical forms, by using tuples, automatas and the operator `in`. The scope of a constraint is given by a tuple of variables on the left of the constraining expression and an automaton is given on the right of the constraining expression. Automatas in PyCSP^3 are objects of Class `Automaton` that are built by calling the following constructor:

```
def __init__(self, *, start, transitions, final):
```

Three named parameters are required:

- `start` is the name of the initial state (a string)
- `transitions` is a set (or list) of 3-tuples
- `final` is the set (or list) of the names of final states (strings)

Note that the set of states and the alphabet can be inferred from `transitions`.



As an example, the constraint defined on scope $\langle x_1, x_2, \ldots, x_7 \rangle$ from the simple automation depicted above is given in PyCSP^3 by:

```
t = {("a",0,"a"), ("a",1,"b"), ("b",1,"c"), ("c",0,"d"),
     ("d",0,"d"), ("d",1,"e"), ("e,0,"e")}
automaton = Automaton(start="a", transitions=t, final="e")

staisfy(
   (x1, x2, x3, x4, x5, x6, x7) in automaton,

   ...
)
```

This gives, after compiling to XCSP^3:

```
<regular>
   <list> x1 x2 x3 x4 x5 x6 x7 </list>
   <transitions>
     (a,0,a)(a,1,b)(b,1,c)(c,0,d)(d,0,d)(d,1,e)(e,0,e)
   </transitions>
   <start> a </start>
   <final> e </final>
</regular>
```

Traveling Tournament with Predefined Venues. This problem was introduced in Section 3.2. Here is a snippet of the PyCSP^3 model:

```
def automaton():
   tr = {("q", 0, "q01"), ("q", 1, "q11"), ("q01", 0, "q02"), ("q01", 1, "q11"),
     ("q11", 0, "q01"), ("q11", 1, "q12"), ("q02", 1, "q11"), ("q12", 0, "q01")}
   return Automaton(start="q", final={"q01", "q02", "q11", "q12"}, transitions=tr)

satisfy(
   # at most 2 consecutive games at home, or consecutive games away
   [h[i] in automaton() for i in range(nTeams)],

   ...
)
```

3.4 Constraint `mdd`

3.5 Constraint `allDifferent`

3.6 Constraint `allEqual`

The constraint `allEqual` ensures that all involved variables take the same value.

Semantics 5

`allEqual`(X)`, with` $X = \langle x_0, x_1, \ldots \rangle$`, iff`
$\forall (i,j) : 0 \leq i < j < |X|, \boldsymbol{x}_i = \boldsymbol{x}_j$

In Python, we can call the function `AllEqual`() with a list of variables as parameter.

Domino. As an illustration, let us consider the problem Domino that was introduced in [17] to emphasize the sub-optimality of a generic constraint propagation algorithm (called AC3). Each instance, characterized by two integers n and d, is binary and corresponds to an undirected constraint graph with a cycle. More precisely, n denotes the number of variables, each with $\{0, \ldots, d-1\}$ as domain, and there exist:

- $n-1$ equality constraints: $x_i = x_{i+1}, \forall i \in \{0, \ldots, n-2\}$
- a trigger constraint: $(x_0 + 1 = x_{n-1}) \vee (x_0 = x_{n-1} = d-1)$

A PyCSP^3 model of this problem is given by the following file 'Domino.py':

PyCSP³ Model 28

```
from pycsp3 import *

n, d = data.nDominos, data.nValues

# x[i] is the value of the ith domino
x = VarArray(size=n, dom=range(d))

satisfy(
   AllEqual(x),

   (x[0] + 1 == x[- 1]) | ((x[0] == x[- 1]) == d - 1)
)
```

Of course, it is possible to replace the constraint `allEqual` by:

```
  [x[i] == x[i + 1] for i in range(n - 1)],
```

The constraint `allEqual` is mainly introduced for its ease of use.

3.7 Constraints `increasing` and `decreasing`

3.8 Constraints `lexIncreasing` and `lexDecreasing`

3.9 Constraint `sum`

3.10 Constraint `count`

3.11 Constraint `nValues`

The constraint `nValues` [3], ensures that the number of distinct values taken by the variables of a specified list X respects a numerical condition $(\odot, k)$. A variant, called `nValuesExcept` [3] discards some specified values of a set E (often, the single value 0).

Semantics 6

`nValues`$(X, E, (\odot, k))$`, with` $X = \langle x_0, x_1, \ldots \rangle$`, iff`
$|\{\boldsymbol{x}_i : 0 \leq i < |X|\} \setminus E| \odot \boldsymbol{k}$
`nValues`$(X, (\odot, k))$ `iff` `nValues`$(X, \emptyset, (\odot, k))$

In PyCSP³, to post a constraint `nValues`, we must call the function `NValues`() whose signature is:

```
  def NValues(term, *others, excepting=None):
```

The two parameters `term` and `others` are positional, and allow us to pass the variables either in sequence (individually) or under the form of a list. The optional named parameter `excepting` allows us to specify a value (integer) or a list of values. The object obtained when calling `NValues`() must be restricted by a condition (typically, defined by a relational operator and a limit).

Board Coloration. This problem was introduced in Section 1.2.2. The constraint `nValues` was introduced for capturing `notAllEqual`.

3.12 Constraint `cardinality`

The constraint `cardinality`, also called `globalCardinality` or `gcc` in the literature, see [15, 10], ensures that the number of occurrences of each value in a specified set V, taken by the variables of a specified list X, is equal to a specified value (or variable), or belongs to a specified interval (information given by a set O). A Boolean option `closed`, when set to `true`, means that all variables of X must be assigned a value from V.

For simplicity, for the semantics below, we assume that V only contains values and O only contains variables. Note that ^{cl} means that `closed` is `true`.

Semantics 7

`cardinality`(X, V, O), `with` $X = \langle x_0, x_1, \ldots \rangle$, $V = \langle v_0, v_1, \ldots \rangle$, $O = \langle o_0, o_1, \ldots \rangle$,
  `iff` $\forall j : 0 \leq j < |V|, |\{i : 0 \leq i < |X| \wedge \boldsymbol{x}_i = v_j\}| = \boldsymbol{o}_j$
`cardinality`$^{cl}(X, V, O)$ `iff` `cardinality`$(X, V, O) \wedge \forall i : 0 \leq i < |X|, \boldsymbol{x}_i \in V$

Prerequisite : $|X| \geq 2 \wedge |V| = |O| \geq 1$

The form of the constraint obtained by only considering variables in the sets X, V and O is called `distribute` in MiniZinc. In that case, for the semantics, me must additionally guarantee:

$$\forall (i, j) : 0 \leq i < j < |V|, \boldsymbol{v}_i \neq \boldsymbol{v}_j.$$

In PyCSP^3, to post a constraint `cardinality`, we must call the function `Cardinality`() whose signature is:

```
def Cardinality(term, *others, occurrences, closed=False):
```

The two parameters `term` and `others` are positional, and allow us to pass the variables either in sequence (individually) or under the form of a list. The value of the required named parameter `occurrences` must be a dictionary: each entry (k, v) in the dictionary means that the number of occurrences of k is given by v. The optionnal named parameter`closed` , when set to `true`, means that all variables specified by the two positional parameters must be assigned a value that corresponds to a key in the dictionnary.

Labeled Dice. From Jim Orlins Blog: "There are 13 words as follows: buoy, cave, celt, flub, fork, hemp, judy, junk, limn, quip, swag, visa, wish. There are 24 different letters that appear in the 13 words. The question is: can one assign the 24 letters to 4 different cubes so that the four letters of each word appears on different cubes. There is one letter from each word on each cube. The puzzle was created by Humphrey Dudley"

A PyCSP^3 model of this problem is given by the following file 'LabeledDice.py':

PyCSP^3 Model 29

```
from pycsp3 import *

words = ["buoy", "cave", "celt", "flub", "fork", "hemp",
         "judy", "junk", "limn", "quip", "swag", "visa"]

# x[i] is the cube where the ith letter of the alphabet is put
x = VarArray(size=26, dom=lambda i: range(1, 5)
       if any(i in to_alphabet_positions(w) for w in words) else None)

satisfy(
   # the four letters of each word appears on different cubes
```

```
    [AllDifferent(x[i] for i in to_alphabet_positions(w)) for w in words],

    # each cube is assigned 6 letters
    Cardinality(x, occurrences={i: 6 for i in range(1, 5)})
)
```

The PyCSP^3 function `to_alphabet_positions()` returns a tuple composed with the position in the alphabet of all letters of a specified string. For example, `to_alphabet_positions("about")` returns `(0, 1, 14, 20, 19)`. The posted `cardinality` constraint ensures that we have 6 letters per cube (using an index i for cubes, ranging from 1 to 4).

Magic Sequence. This problem was introduced in Section 1.2.3. Here is a snippet of the PyCSP^3 model:

```
# x[i] is the ith value of the sequence
x = VarArray(size=n, dom=range(n))

satisfy(
   # each value i occurs exactly x[i] times in the sequence
   Cardinality(x, occurrences={i: x[i] for i in range(n)}),

   ...
)
```

Here, one can see that variables are used for counting the number of occurrences, and besides, this is a special case where these variables are from the main list (first parameter x).

Sports Scheduling. From CSPLib: "The problem is to schedule a tournament of n teams over $n-1$ weeks, with each week divided into $n/2$ periods, and each period divided into two slots. The first team in each slot plays at home, whilst the second plays the first team away. A tournament must satisfy the following three constraints:

- every team plays once a week;
- every team plays at most twice in the same period over the tournament;
- every team plays every other team.

"

A PyCSP^3 model of this problem is given by the following file 'SportsScheduling.py':

PyCSP^3 Model 30

```
from pycsp3 import *

nTeams = data.nTeams
nWeeks, nPeriods = nTeams - 1, nTeams // 2
nMatches = (nTeams - 1) * nTeams // 2

def match_number(t1, t2):
   return nMatches - ((nTeams - t1) * (nTeams - t1 - 1)) // 2 + (t2 - t1 - 1)

table = {(t1, t2, match_number(t1, t2)) for t1 in range(nTeams)
                                        for t2 in range(t1 + 1, nTeams)}

# h[w][p] is the home team at week w and period p
h = VarArray(size=[nWeeks, nPeriods], dom=range(nTeams))

# a[w][p] is the away team at week w and period p
a = VarArray(size=[nWeeks, nPeriods], dom=range(nTeams))
```

```
# m[w][p] is the number of the match at week w and period p
m = VarArray(size=[nWeeks, nPeriods], dom=range(nMatches))

satisfy(
   # linking variables through ternary table constraints
   [(h[w][p], a[w][p], m[w][p]) in table for w in range(nWeeks)
                                        for p in range(nPeriods)],

   # all matches are different (no team can play twice against another team)
   AllDifferent(m),

   # each week, all teams are different (each team plays each week)
   [AllDifferent(h[w] + a[w]) for w in range(nWeeks)],

   # each team plays at most two times in each period
   [Cardinality(h[:, p] + a[:, p], occurrences={t: range(1, 3)
       for t in range(nTeams)}) for p in range(nPeriods)],
)
```

Here, we can see that the interval 1..2 (given by `range(1,3)`) is used to control the number of occurrences of each team in each period, when posting `cardinality` constraints. Note that we could add some symmetry breaking constraints to the model.

3.13 Constraint `maximum`

The constraint `maximum` ensures that the maximum value among those assigned to the variables of a specified list X respects a numerical condition $(\odot, k)$.

Semantics 8

`maximum`$(X, (\odot, k))$, `with` $X = \langle x_0, x_1, \ldots \rangle$, `iff`
$\max\{\boldsymbol{x}_i : 0 \leq i < |X|\} \odot \boldsymbol{k}$

In PyCSP^3, to post a constraint `maximum`, we must call the function `Maximum`() whose signature is:

```
def Maximum(term, *others)
```

The two parameters `term` and `others` are positional, and allow us to pass the variables either in sequence (individually) or under the form of a list. The object obtained when calling `Maximum`() must be restricted by a condition (typically, defined by a relational operator and a limit).

Open Stacks. From Steven Prestwich: "A manufacturer has a number of orders from customers to satisfy. Each order is for a number of different products, and only one product can be made at a time. Once a customers order is started a stack is created for that customer. When all the products that a customer requires have been made the order is sent to the customer, so that the stack is closed. Because of limited space in the production area, the number of stacks that are simultaneously open should be minimized."

An example of data is given by the following JSON file:

```
{
  "orders": [
    [0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,0],
    [1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    ...
  ]
```

```
}
```

Each row of `orders` corresponds to a customer order indicating with 0 or 1 if the jth product is needed. A PyCSP^3 model of this problem is given by the following file 'OpenStacks.py':

PyCSP^3 Model 31

```
from pycsp3 import *

orders = data.orders
n = len(orders))    # n orders (customers)
m = len(orders[0])  # m possible products

def table(t):
   return {(ANY, te, 0) for te in range(t)} |
          {(ts, ANY, 0) for ts in range(t + 1, m)} |
          {(ts, te, 1) for ts in range(t + 1) for te in range(t, m)}

# p[j] is the period (time) of the jth product
p = VarArray(size=m, dom=range(m))

# s[i] is the starting time of the ith stack
s = VarArray(size=n, dom=range(m))

# e[i] is the ending time of the ith stack
e = VarArray(size=n, dom=range(m))

# o[i][t] is 1 iff the ith stack is open at time t
o = VarArray(size=[n, m], dom={0, 1})

# ns[t] is the number of open stacks at time t
ns = VarArray(size=m, dom=range(m + 1))

satisfy(
   # all products are scheduled at different times
   AllDifferent(p),

   # computing starting times of stacks
   [Minimum(p[j] for j in range(m) if orders[i][j] == 1) == s[i] for i in range(n)],

   # computing ending times of stacks
   [Maximum(p[j] for j in range(m) if orders[i][j] == 1) == e[i] for i in range(n)],

   # inferring when stacks are open
   [(s[i], e[i], o[i][t]) in table(t) for i in range(n) for t in range(m)],

   # computing the number of open stacks at any time
   [Sum(o[:, t]) == ns[t] for t in range(m)]
)

minimize(
   # minimizing the number of stacks that are simultaneously open
   Maximum(ns)
)
```

Note that each list of variables is given to `Maximum`() under the form of a comprehension list (generator). The PyCSP^3 function `Maximum`() is also used for building the expression to be minimized.

3.14 Constraint `minimum`

The constraint `minimum` ensures that the minimum value among those assigned to the variables of a specified list X respects a numerical condition $(\odot, k)$.

Semantics 9

`minimum`$(X, (\odot, k))$`, with` $X = \langle x_0, x_1, \ldots \rangle$`, iff`
$\min\{\boldsymbol{x}_i : 0 \leq i < |X|\} \odot \boldsymbol{k}$

In PyCSP^3, to post a constraint `minimum`, we must call the function `Minimum`() whose signature is:

```
def Minimum(term, *others)
```

The two parameters `term` and `others` are positional, and allow us to pass the variables either in sequence (individually) or under the form of a list. The object obtained when calling `Minimum`() must be restricted by a condition (typically, defined by a relational operator and a limit).

Open Stacks. See the model introduced in the previous section.

3.15 Constraint `element`

The constraint `element` [9] ensures that the element of a specified list X at a specified index i has a specified value v. The semantics is `X[i] = v`, or equivalently:

Semantics 10

`element`(X, i, v)`, with` $X = \langle x_0, x_1, \ldots \rangle$`, iff`
$\boldsymbol{x_i} = \boldsymbol{v}$

It is important to note that i must be an integer variable (and not a constant). In Python, to post an `element` constraint, we use the facilities offered by the language, meaning that we can write expressions involving relational and indexing ([]) operators.

There are three variants of `element`:

- variant 1: X is a list of variables, i is an integer variable and v is an integer variable
- variant 2: X is a list of variables, i is an integer variable and v is an integer (constant)
- variant 3: X is a list of integers, i is an integer variable and v is an integer variable

Although the variant 3 can be reformulated as a binary extensional constraint, it is often used when modeling.

The Sandwich case. From beCool (UCLouvain): "Someone in the university ate Alice's sandwich at the cafeteria. We want to find out who the culprit is. The witnesses are unanimous about the following facts:

1. Three persons were in the cafeteria at the time of the crime: Alice, Bob, and Sascha.
2. The culprit likes Alice.
3. The culprit is taller than Alice.
4. Nobody is taller than himself.

5. If A is taller than B, then B is not taller than A.
6. Bob likes no one that Alice likes.
7. Alice likes everybody except Bob.
8. Sascha likes everyone that Alice likes.
9. Nobody likes everyone.

”

This is a single problem (no external data is required). A PyCSP^3 model of this problem is given by the following file 'Sandwich.py':

PyCSP^3 Model 32

```
from pycsp3 import *

alice, bob, sascha = 0, 1, 2

# culprit is among alice (0), bob (1) and sascha (2)
culprit = Var(dom=range(3))

# likes[i][j] is 1 iff the ith guy likes the jth guy
likes = VarArray(size=[3, 3], dom={0, 1})

# taller[i][j] is 1 iff the ith guy is taller than the jth guy
taller = VarArray(size=[3, 3], dom={0, 1})

satisfy(
    # the culprit likes Alice
    likes[culprit][alice] == 1,

    # the culprit is taller than Alice
    taller[culprit][alice] == 1,

    # nobody is taller than himself
    [taller[i][i] == 0 for i in range(3)],

    # the ith guy is taller than the jth guy iff the reverse is not true
    [taller[i][j] != taller[j][i] for i in range(3) for j in range(3) if i != j],

    # Bob likes no one that Alice likes
    [imply(likes[alice][i], ~likes[bob][i]) for i in range(3)],

    # Alice likes everybody except Bob
    [likes[alice][i] == 1 for i in range(3) if i != bob],

    # Sascha likes everyone that Alice likes
    [imply(likes[alice][i], likes[sascha][i]) for i in range(3)],

    # nobody likes everyone
    [Count(likes[i], value=0) >= 1 for i in range(3)]
)
```

The variant 2 of `element` is illustrated by:

```
likes[culprit][alice] == 1,
```

as it basically encodes “the variable at index `culprit` in the column 0 (alice) of the 2-dimensional array of variables `likes` must be equal to 1”.

Warehouse Location. This problem was introduced in Section 1.3.2. Here is a snippet of the PyCSP^3 model:

```
satisfy(
    ...

    # computing the cost of supplying the ith store
    [costs[i][w[i]] == c[i] for i in range(nStores)]
)
```

The variant 3 of `element` is illustrated by:

```
costs[i][w[i]] == c[i]
```

as it basically encodes "the variable at index `w[i]` in the ith row of the 2-dimensional array of integers `costs` must be equal to c[i]".

It is also possible to use a variant of `element` on matrices, i.e., by using two indexes given by integer variables.

3.16 Constraint `channel`

The first variant of the constraint `channel` is defined on a single list of variables, and ensures that if the ith variable of the list is assigned the value j, then the jth variable of the same list must be assigned the value i.

Semantics 11

`channel`(X), `with` $X = \langle x_0, x_1, \ldots \rangle$, `iff`
$\forall i : 0 \leq i < |X|, \boldsymbol{x}_i = j \Rightarrow \boldsymbol{x}_j = i$

A second classical variant of `channel`, sometimes called `inverse` or `assignment` in the literature, is defined from two separate lists (of the same size) of variables. It ensures that the value assigned to the ith variable of the first list gives the position of the variable of the second list that is assigned to i, and vice versa.

Semantics 12

`channel`(X, Y), `with` $X = \langle x_0, x_1, \ldots \rangle$ `and` $Y = \langle y_0, y_1, \ldots \rangle$, `iff`
$\forall i : 0 \leq i < |X|, \boldsymbol{x}_i = j \Leftrightarrow \boldsymbol{y}_j = i$

Prerequisite: $2 \leq |X| = |Y|$

It is also possible to use this form of `channel`, with two lists of different sizes. The constraint then imposes restrictions on all variables of the first list, but not on all variables of the second list. The syntax is the same, but the semantics is the following (note that the equivalence has been replaced by an implication):

Semantics 13

`channel`(X, Y), `with` $X = \langle x_0, x_1, \ldots \rangle$ `and` $Y = \langle y_0, y_1, \ldots \rangle$, `iff`
$\forall i : 0 \leq i < |X|, \boldsymbol{x}_i = j \Rightarrow \boldsymbol{y}_j = i$

Prerequisite: $2 \leq |X| < |Y|$

Finally, a third variant of `channel` is obtained by considering a list of 0/1 variables to be channeled with an integer variable. This third form of constraint `channel` ensures that the only variable of

the list that is assigned to 1 is at an index (position) that corresponds to the value assigned to the stand-alone integer variable.

Semantics 14

`channel`(X, v)`, with` $X = \{x_0, x_1, \ldots\}$`, iff`
$\forall i : 0 \leq i < |X|, \boldsymbol{x}_i = 1 \Leftrightarrow \boldsymbol{v} = i$
$\exists i : 0 \leq i < |X| \wedge \boldsymbol{x}_i = 1$

In PyCSP^3, to post a constraint `channel`, we must call the function `Channel`() whose signature is:

```
def Channel(list1, list2=None, *, start_index1=0, start_index2=0):
```

For the first variant, in addition to the positional parameter `list1`, one may use the the optional attribute `start_index1` that gives the number used for indexing the first variable in this list (0, by default). For the second variant, two lists must be specified, and optionally the two named parameters can be used. For the third variant, the positional parameter `list2` must be a variable (or a list only containing one variable).

Blackhole. This problem was introduced in Section 1.3.3. Here is a snippet of the PyCSP^3 model:

```
...

# x[i] is the value j of the card at position i of the stack
x = VarArray(size=nCards, dom=range(nCards))

# y[j] is the position i of the card whose value is j
y = VarArray(size=nCards, dom=range(nCards))

satisfy(
   Channel(x, y),

   ...
)
```

The constraint `channel` (second variant) links the dual roles of variables from arrays x and y.

Progressive Party Problem. From CSPLib: "The problem is to timetable a party at a yacht club. Certain boats are to be designated hosts, and the crews of the remaining boats in turn visit the host boats for several successive half-hour periods. The crew of a host boat remains on board to act as hosts while the crew of a guest boat together visits several hosts. Every boat can only hold a limited number of people at a time (its capacity) and crew sizes are different. The total number of people aboard a boat, including the host crew and guest crews, must not exceed the capacity. A guest boat cannot revisit a host and guest crews cannot meet more than once. The problem facing the rally organizer is that of minimizing the number of host boats."

An example of data is given by the following JSON file:

```
{
  "nPeriods": 5,
  "boats": [
    {"crewSize": 2, "capacity": 6},
    {"crewSize": 2, "capacity": 8},
    ...
  ]
}
```

A PyCSP^3 model of this problem is given by the following file 'ProgressiveParty.py':

PyCSP³ Model 33

```
from pycsp3 import *

nPeriods = data.nPeriods
boats, nBoats = data.boats, len(data.boats)
crews = [boat.crewSize for boat in boats]

# h[b] indicates if the boat b is a host boat
h = VarArray(size=nBoats, dom={0, 1})

# s[b][p] is the scheduled (visited) boat by the crew of boat b at period p
s = VarArray(size=[nBoats, nPeriods], dom=range(nBoats))

# g[b1][p][b2] is 1 if s[b1][p] = b2
g = VarArray(size=[nBoats, nPeriods, nBoats], dom={0, 1})

satisfy(
   # identifying host boats (when receiving)
   [iff(s[b][p] == b, h[b]) for b in range(nBoats) for p in range(nPeriods)],

   # identifying host boats (when visiting)
   [imply(s[b1][p] == b2, h[b2]) for b1 in range(nBoats) for b2 in range(nBoats)
     if b1 != b2 for p in range(nPeriods)],

   # channeling variables from arrays s and g
   [Channel(g[b][p], s[b][p]) for b in range(nBoats) for p in range(nPeriods)],

   # boat capacities must be respected
   [g[:, p, b] * crews <= boats[b].capacity for b in range(nBoats)
     for p in range(nPeriods)],

   # a guest crew cannot revisit a host
   [AllDifferent(s[b], excepting=b) for b in range(nBoats)],

   # guest crews cannot meet more than once
   [Sum(s[b1][p] == s[b2][p] for p in range(nPeriods)) <= 2
     for b1 in range(nBoats) for b2 in range(b1 + 1, nBoats)]
}

minimize(
    # minimizing the number of host boats
    Sum(h)
)
```

This is the third variant of `channel` that is used here: `g[b][p]` is an array of 0/1 variables while `s[b][p]` is a stand-alone integer variable. Below, note how the symbol ':' is used to take a complete slice of a 3-dimensional array of variables, when posting constraints about boat capacities. Instead, we could have written:

```
[[g[i][p][b] for i in range(nBoats)] * crews <= boats[b].capacity
  for b in range(nBoats) for p in range(nPeriods)],
```

Concerning the last list of `sum` constraints, as the Boolean expression `s[b1][p] == s[b2][p]` is considered to return integers, 0 for false and 1 for true, it is possible to perform a summation.

3.17 Constraint `noOverlap`

We start with the one dimensional form of `noOverlap` [10] that corresponds to `disjunctive` [5] and ensures that some objects (e.g., tasks), defined by their origins (e.g., starting times) and lengths (e.g., durations), must not overlap. The semantics is given by:

Semantics 15

`noOverlap`(X, L), `with` $X = \langle x_0, x_1, \ldots \rangle$ `and` $L = \langle l_0, l_1, \ldots \rangle$, `iff`
$\forall (i, j) : 0 \leq i < j < |X|, \boldsymbol{x}_i + \boldsymbol{l}_i \leq \boldsymbol{x}_j \vee \boldsymbol{x}_j + \boldsymbol{l}_j \leq \boldsymbol{x}_i$

Prerequisite : $|X| = |L| \geq 2$

In PyCSP^3, to post a constraint `noOverlap`, we must call the function `NoOverlap`() whose signature is:

```
def NoOverlap(*, origins, lengths, zero_ignored=False):
```

Note that all parameters must be named (see '*' at first position), and that the parameter `zero_ignored` is optional (value `False` by default). If ever we are in a situation where there exist some zero-length object(s), then if the parameter `zero_ignored` is set to `False`, it indicates that zero-length objects cannot be packed anywhere (cannot overlap with other objects). Arguments given to `origins` and `lengths` when calling the function `NoOverlap`() are expected to be lists of the same length; `origins` must be given a list of variables whereas `lengths` must be given either a list of variables or a list of integers.

Flow Shop Scheduling. Frow WikiPedia: "There are n machines and m jobs. Each job contains exactly n operations. The ith operation of the job must be executed on the ith machine. No machine can perform more than one operation simultaneously. For each operation of each job, execution time is specified. Operations within one job must be performed in the specified order. The first operation gets executed on the first machine, then (as the first operation is finished) the second operation on the second machine, and so on until the nth operation. Jobs can be executed in any order, however. Problem definition implies that this job order is exactly the same for each machine. The problem is to determine the optimal such arrangement, i.e. the one with the shortest possible total job execution makespan."

To specify a problem instance, we just need a two-dimensional array of integers for recording durations, as in the following JSON file:

```
{
  "durations":[
    [26,59,78,88,69],
    [38,62,90,54,30],
    ...
  ]
}
```

A PyCSP^3 model of this problem is given by the following file 'FlowShopScheduling.py':

PyCSP^3 Model 34

```
from pycsp3 import *

durations = data.durations  # durations[i][j] is the duration of op. j for job i
n, m = len(durations), len(durations[0])
horizon = sum(sum(t) for t in durations) + 1

# s[i][j] is the start time of the jth operation for the ith job
s = VarArray(size=[n, m], dom=range(horizon))

# e[i] is the end time of the last operation for the ith job
e = VarArray(size=n, dom=range(horizon))

satisfy(
```

```
    # operations must be ordered on each job
    [Increasing(s[i], lengths=durations[i]) for i in range(n)],

    # computing the end time of each job
    [e[i] == s[i][- 1] + durations[i][- 1] for i in range(n)],

    # no overlap on resources
    [NoOverlap(origins=s[:, j], lengths=durations[:, j]) for j in range(m)]
)

minimize(
    # minimizing the makespan
    Maximum(e)
)
```

In this model, for each operation (or equivalently, machine) j, we collect the list of variables from the jth column of `s` and the list of integers from the jth column of `durations` when posting a constraint `noOverlap`. Remember that the notation `[:, j]` stands for the jth column of a two-dimensional array (list).

The k-dimensional form of `noOverlap` corresponds to `diffn` [2] and ensures that, given a set of n-dimensional boxes; for any pair of such boxes, there exists at least one dimension where one box is after the other, i.e., the boxes do not overlap. The semantics is:

Semantics 16

`noOverlap`$(\mathcal{X}, \mathcal{L})$, `with` $\mathcal{X} = \langle (x_{1,1}, \ldots, x_{1,n}), (x_{2,1}, \ldots, x_{2,n}), \ldots \rangle$ `and`
$\mathcal{L} = \langle (l_{1,1}, \ldots, l_{1,n}), (l_{2,1}, \ldots, l_{2,n}), \ldots \rangle$, `iff`
$\forall (i,j) : 1 \leq i < j \leq |\mathcal{X}|, \exists k \in 1..n : \boldsymbol{x}_{i,k} + \boldsymbol{l}_{i,k} \leq \boldsymbol{x}_{j,k} \vee \boldsymbol{x}_{j,k} + \boldsymbol{l}_{j,k} \leq \boldsymbol{x}_{i,k}$

Prerequisite : $|\mathcal{X}| = |\mathcal{L}| \geq 2$

In PyCSP^3, to post a constraint `noOverlap`, we must call the function `NoOverlap`() whose signature is:

```
def NoOverlap(*, origins, lengths, zero_ignored=False):
```

Note that all parameters must be named (see '*' at first position), and that the parameter `zero_ignored` is optional (value `False` by default). If ever we are in a situation where there exist some zero-length box(es), then if the parameter `zero_ignored` is set to `False`, it indicates that zero-length boxes cannot be packed anywhere (cannot overlap with other boxes). Arguments given to `origins` and `lengths` when calling the function `NoOverlap`() are expected to be two-dimensional lists of the same length; `origins` must only involve variables whereas `lengths` must involve either only variables or only integers.

Rectangle packing problem. The rectangle packing problem consists of finding a way of putting a given set of rectangles (boxes) in an enclosing rectangle (container) without overlap.

An example of data is given by the following JSON file:

```
{
  "container":{"width":112,"height":112},
  "boxes":[
    {"width":2,"height":2},
    {"width":4,"height":4},
    ...
  ]
}
```

A PyCSP^3 model of this problem is given by the following file 'RectanglePacking.py':

PyCSP^3 Model 35

```
from pycsp3 import *

width, height = data.container.width, data.container.height
boxes = data.boxes
nBoxes = len(boxes)

# x[i] is the x-coordinate where is put the ith box (rectangle)
x = VarArray(size=nBoxes, dom=range(width))

# y[i] is the y-coordinate where is put the ith box (rectangle)
y = VarArray(size=nBoxes, dom=range(height))

satisfy(
    # unary constraints on x
    [x[i] + boxes[i].width <= width for i in range(nBoxes)],

    # unary constraints on y
    [y[i] + boxes[i].height <= height for i in range(nBoxes)],

    # no overlap on boxes
    NoOverlap(origins=[(x[i], y[i]) for i in range(nBoxes)],
              lengths=[(box.width, box.height) for box in boxes]),

    # tag(symmetry-breaking)
    [
      x[-1] <= math.floor((width - boxes[-1].width) // 2.0),
      y[-1] <= x[-1]
    ] if width == height else None
)
```

3.18 Constraint `Cumulative`

The constraint `cumulative` is useful when a resource of limited quantity must be shared for achieving several tasks. So, the context is to manage a collection of tasks, each one being described by 4 attributes: its starting time `origin`, its length or duration `length`, its stopping time `end` and its resource consumption `height`. Usually, the values for `length` and `height` are given while the values for `origin` (and `end` by deduction) must be computed.

The constraint `cumulative` [1] enforces that at each point in time, the cumulated height of tasks that overlap that point, respects a numerical condition $(\odot, k)$. The semantics is given by:

Semantics 17

`cumulative`$(X, L, H, (\odot, k))$, `with` $X = \langle x_0, x_1, \ldots \rangle$, $L = \langle l_0, l_1, \ldots \rangle$, $H = \langle h_0, h_1, \ldots \rangle$, `iff`

$\forall t \in \mathbb{N}, \sum \{\boldsymbol{h}_i : 0 \leq i < |H| \wedge \boldsymbol{x}_i \leq t < \boldsymbol{x}_i + \boldsymbol{l}_i\} \odot \boldsymbol{k}$

Prerequisite : $|X| = |L| = |H| \geq 2$

If the attributes `end` are present while reasoning, we have additionally a set $E = \langle e_0, e_1, \ldots \rangle$ such that:

$$\forall i : 0 \leq i < |X|, \boldsymbol{x}_i + \boldsymbol{l}_i = \boldsymbol{e}_i$$

In PyCSP^3, to post a constraint `cumulative`, we must call the function `Cumulative()` whose signature is:

```
def Cumulative(*, origins, lengths, heights, ends=None):
```

Note that all parameters must be named (see '*' at first position) and the parameter `ends` is optional (value `None` by default). Arguments given when calling the function are expected to be lists of the same length. The object obtained when calling `Cumulative()` must be restricted by a condition (typically, defined by a relational operator and a limit).

Rcpsp. From CSPLib: "The resource-constrained project scheduling problem is a classical problem in operations research. A number of activities are to be scheduled. Each activity has a duration and cannot be interrupted. There are a set of precedence relations between pairs of activities which state that the second activity must start after the first has finished. There are a set of renewable resources. Each resource has a maximum capacity and at any given time slot no more than this amount can be in use. Each activity has a demand (possibly zero) on each resource. The problem is usually stated as an optimisation problem where the makespan (i.e., the completion time of the last activity) is minimised." See CSPLib–Problem 061 for more information.

An example of data is given by the following JSON file:

```
{
  "horizon":158,
  "resourceCapacities":[12,13,4,12],
  "jobs":[
    {"duration":0, "successors":[1,2,3], "requiredQuantities":[0,0,0,0]},
    {"duration":8, "successors":[5,10,14], "requiredQuantities":[4,0,0,0]},
    ...
  ]
}
```

A PyCSP^3 model of this problem is given by the following file 'Rcpsp.py':

PyCSP^3 Model 36

```
from pycsp3 import *

horizon = data.horizon
capacities = data.resourceCapacities
jobs, nJobs = data.jobs, len(data.jobs)

def cumulative_for(k):
   indexes = [i for i in range(nJobs) if jobs[i].requiredQuantities[k] > 0]
   origins = [s[i] for i in indexes]
   lengths = [jobs[i].duration for i in indexes]
   heights = [jobs[i].requiredQuantities[k] for i in indexes]
   return Cumulative(origins=origins, lengths=lengths, heights=heights)

# s[i] is the starting time of the ith job
s = VarArray(size=nJobs, dom=lambda i: {0} if i == 0 else range(horizon))

satisfy(
   # precedence constraints
   [s[i] + job.duration <= s[j] for i, job in enumerate(jobs)
                                for j in job.successors],

   # resource constraints
   [cumulative_for(k) <= capacity for k, capacity in enumerate(capacities)]
)

minimize(
   s[-1]
```

```
)
```

Observe how the `Cumulative` object returned by the local function call `cumulative_for`(k) is imposed to be less than or equal to the capacity of the kth resource.

3.19 Constraint `Circuit`

Sometimes, problems involve graphs that are defined with integer variables (encoding called "successors variables"). In that context, graph-based constraints, like `circuit`, involve a main list of variables $x_0, x_1, \ldots$ The assumption is that each pair $(i, \boldsymbol{x}_i)$ represents an arc (or edge) of the graph to be built; if $\boldsymbol{x}_i = j$, then it means that the successor of node i is node j. Note that a *loop* (also called self-loop) corresponds to a variable x_i such that $\boldsymbol{x}_i = i$.

The constraint `circuit` [2] ensures that the values taken by the variables of the specified list forms a circuit, with the assumption that each pair $(i, \boldsymbol{x}_i)$ represents an arc. It is also possible to indicates that the circuit must be of a given size (strictly greater than 1). The semantics is given by:

Semantics 18

`circuit`(X)`, with` $X = \langle x_0, x_1, \ldots \rangle$`, iff` `// capture subscircuit`
$\{(i, \boldsymbol{x}_i) : 0 \leq i < |X| \wedge i \neq \boldsymbol{x}_i\}$ `forms a circuit of size` > 1
`circuit`(X, s)`, with` $X = \langle x_0, x_1, \ldots \rangle$`, iff`
$\{(i, \boldsymbol{x}_i) : 0 \leq i < |X| \wedge i \neq \boldsymbol{x}_i\}$ `forms a circuit of size` $\boldsymbol{s} > 1$

In PyCSP^3, to post a constraint `circuit`, we must call the function `Circuit`() whose signature is:

```
def Circuit(term, *others, start_index=0, size=None):
```

The two first parameters `term` and `others` are positional, and allow us to pass the "successors variables" either in sequence (individually) or under the form of a list. The two other parameters are optional (and must be named): `start_index` gives the number used for indexing the first variable of the specified list (0, by default), and `size` indicates that the circuit must be of a given size (`None` by default indicates that no specific size is required).

It is important to note that the circuit is not required to cover all nodes (the nodes that are not present in the circuit are then self-looping). Hence `circuit`, with loops being simply ignored, basically represents `subcircuit` (e.g., in MiniZinc). If ever you need a full circuit (i.e., without any loop), you have three solutions:

- indicate with `size` the number of successor variables
- initially define the variables without the self-looping values,
- post unary constraints.

Mario. From Amaury Ollagnier and Jean-Guillaume Fages, in the context of the 2013 Minizinc Competition: "This models a routing problem based on a little example of Mario's day. Mario is an Italian Plumber and his work is mainly to find gold in the plumbing of all the houses of the neighborhood. Mario is moving in the city using his kart that has a specified amount of fuel. Mario starts his day of work from his house and always ends to his friend Luigi's house to have the supper. The problem here is to plan the best path for Mario in order to earn the more money with the amount of fuel of his kart. From a more general point of view, the problem is to find a path in a graph:

- path endpoints are given (from Mario's to Luigi's)
- the sum of weights associated to arcs in the path is restricted (fuel consumption)
- the sum of weights associated to nodes in the path has to be maximized (gold coins)

"

An example of data is given by the following JSON file:

```
{
  "marioHouse": 0,
  "luigiHouse": 1,
  "fuelLimit": 2000,
  "houses":[
    {
      "fuelConsumption": [0,221,274,80,13,677,670,921,93,969,13,18,217,86,322],
      "gold":0
    },
    {
      "fuelConsumption":[0,0,702,83,813,679,906,246,35,529,79,528,451,242,712],
      "gold":0
    },
    ...
  ]
}
```

A PyCSP^3 model of this problem is given by the following file 'Mario.py':

PyCSP^3 Model 37

```
from pycsp3 import *

# similar model proposed by Ollagnier and Fages for the 2013 Minizinc Competition

marioHouse, luigiHouse = data.marioHouse, data.luigiHouse
fuelLimit = data.fuelLimit
houses, nHouses = data.houses, len(data.houses)
fuels = [house.fuelConsumption for house in houses]

# s[i] is the house succeeding to the ith house (itself if not part of the route)
s = VarArray(size=nHouses, dom=range(nHouses))

# f[i] is the fuel consumed at each step (from house i to its successor)
f = VarArray(size=nHouses, dom=lambda i: set(fuels[i]))

# g[i] is the gold earned at house i
g = VarArray(size=nHouses, dom=lambda i: {0, houses[i].gold})


satisfy(
   # linking variables from arrays s and f
   [fuels[i][s[i]] == f[i] for i in range(nHouses)],

   # we cannot consume more than the available fuel
   Sum(f) <= fuelLimit,

   # gold earned at each house
   [iff(s[i] == i, g[i] == 0)
     for i in range(nHouses) if i not in {marioHouse, luigiHouse}],

   # Mario must make a tour (not necessarily complete)
   Circuit(s),

   # Mario's house succeedes to Luigi's house
```

```
    s[luigiHouse] == marioHouse
)

maximize(
    # maximizing collected gold
    Sum(g)
)
```

For linking variables from arrays s and f, we use some `element` constraints. Note how lists `fuels[i]` involved in these constraints can be directly indexed by variables (objects). This is because the type of `fuels[i]` is a PyCSP^3 subclass of 'list'; and this is automatically handled when loading the JSON file. Suppose that we hould have written instead:

```
fuels = [[v for v in house.fuelConsumption] for house in houses]
```

Here, `fuels[i]` would be a simple 'list', and we would get an error when compiling. In that case, to fix the problem, it is possible to call the PyCSP^3 function `cp_array`():

```
fuels = [cp_array(v for v in house.fuelConsumption) for house in houses]
```

but of course, the code we have chosen for our model above is simpler.

When the list (vector) involved in a constraint `element` contains integers, one may decide to use an `extension` constraint. For our model, this would give:

```
[(s[i], f[i]) in {(j, fuel) for j, fuel in enumerate(fuels[i])} for i in range(nHouses)],
```

Chapter 4

Frequently Asked Questions

This chapter will contain frequently asked questions.

Q. Is it possible to post a constraint only if a condition holds?
A. Of course, it is always possible to put the condition outside the PyCSP^3 function `satisfy()`. For example:

```
if test > 0:
   satisfy(AllDifferent(w, x, y, z))
```

but it is also possible to use the Python conditional operator 'if else' while returning 'None' if the condition does not hold.

```
   satisfy(AllDifferent(w, x, y, z) if test > 0 else None)
```

Bibliography

[1] A. Aggoun and N. Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[2] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[3] C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. Filtering algorithms for the nvalue constraint. *Constraints*, 11(4):271–293, 2006.

[4] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette. XCSP3: An integrated format for benchmarking combinatorial constrained problems. Technical Report arXiv:1611.03398, Specifications 3.0.5, CoRR, 2016-2017. Available from http://www.xcsp.org/format3.pdf.

[5] J. Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11:42–47, 1982.

[6] M. Carlsson and N. Beldiceanu. From constraints to finite automata to filtering algorithms. In *Proceedings of ESOP'04*, pages 94–108, 2004.

[7] T. Guns. Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation, held with CP'19*, 2019.

[8] E. Hebrard, E. O'Mahony, and B. O'Sullivan. Constraint programming and combinatorial optimisation in Numberjack. In *Proceedings of CPAIOR'10*, pages 181–185, 2010.

[9] P. Van Hentenryck and J.-P. Carillon. Generality versus specificity: An experience with AI and OR techniques. In *Proceedings of AAAI'88*, pages 660–664, 1988.

[10] J.N. Hooker. *Integrated Methods for Optimization*. Springer, 2012.

[11] Christopher Jefferson and Peter Nightingale. Extending simple tabular reduction with short supports. In *Proceedings of IJCAI'13*, pages 573–579, 2013.

[12] C. Lecoutre. JvCSP^3: A java API for modeling constrained combinatorial problems (version 1.1). Technical report, CRIL, 2018.

[13] R. Melo, S. Urrutia, and C. Ribeiro. The traveling tournament problem with predefined venues. *Journal of Scheduling*, 12(6):607–622, 2009.

[14] G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'04*, pages 482–495, 2004.

[15] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of AAAI'96*, pages 209–215, 1996.

[16] H. Verhaeghe, C. Lecoutre, and P. Schaus. Extending compact-table to negative and short tables. In *Proceedings of AAAI'17*, pages 3951–3957, 2017.

[17] Y. Zhang and R. Yap. Making AC3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321, 2001.