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

Version 1.1 – May 10, 2020

https://github.com/xcsp3team/pycsp3

Abstract

In this document, we introduce PyCSP^3, a Python library that allows us to write models of combinatorial constrained problems in a simple and declarative way. Currently, with PyCSP^3, you can write models of constraint 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) to generate an XCSP^3 instance (file), and you solve that problem instance by means of a constraint solver. In this document, you will find all that you need to know about PyCSP^3, with more than 40 illustrative models.

In a nutshell, 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

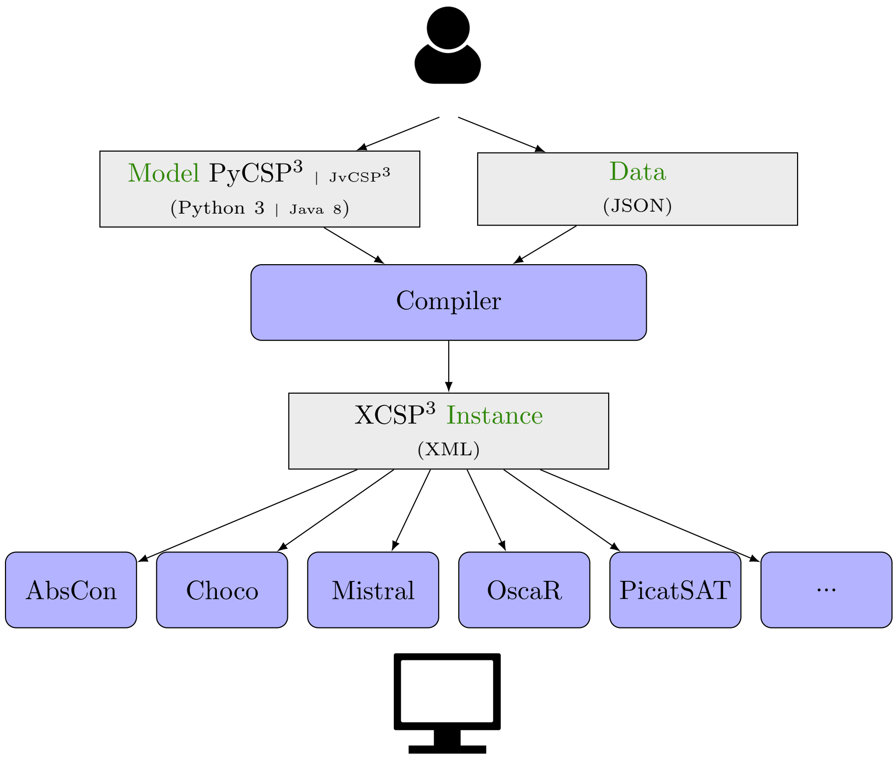


Figure 1: Complete process for modeling and solving combinatorial constrained problems.

For modeling, as indicated above, the user can choose between two well-known languages (Python or 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 a constraint solver as, e.g., AbsCon, Choco, OscaR or 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. Note that using JSON instead of XML for representing instances would have been possible but has some drawbacks, as explained in an appendix of XCSP^3 Specifications [6].

Licence. PyCSP^3 is licensed under the MIT License

Code. PyCSP^3 code is available

- on Github: https://github.com/xcsp3team/pycsp3
- as a PyPi package: https://pypi.org/project/pycsp3

Finally, note that PyCSP^3 is inspired from both JvCSP^3 [25] and Numberjack [20]. It has also connections with CPpy [19].

Contents

Chapter 1

Illustrative Models in PyCSP^3

Warning. In this chapter, we gently 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. We call them *single* problems because they are unique (meaning that we do not need to provide any external data when compiling them).

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 four successive integer numbers sum up to 14?

Figure 1.1: Famous riddles in Carambar candies. (image from www.flickr.com)

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 values that can be assigned to the integer variables because, by using that interval of values, we are absolutely certain of not losing any solutions while avoiding to reason with an infinite set 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 1 *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 (and even possibly in a list, although not shown here) 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 generators and comprehension lists/sets. 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 still 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 Table 1.1, you can find a few other examples of `intension` constraints, while in Tables 1.2 and 1.3, you can find the available operators and functions in PyCSP^3.

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 generated 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>
```

Expressions	Observations
$x + y < 10$	equivalent to $10 > x + y$
$x * 2 - 10 * y + 5 == 100$	we need to use '==' in Python
$\texttt{abs}(z[0] - z[1]) >= 2$	equivalent to $\texttt{dist}(z[0], z[1]) >= 2$
$(x == y) \mid (y == 0)$	parentheses are required
$\texttt{disjunction}(x < 2, y < 4, x > y)$	equivalent to $(x < 2) \mid (y < 4) \mid (x > y)$
$\texttt{imply}(x == 0, y > 0)$	equivalent to $(x \mathrel{!}= 0) \mid (y > 0)$
$\texttt{iff}(x > 0, y > 0)$	equivalent to $(x > 0) == (y > 0)$
$(x == 0)$ ^ $(y == 1)$	use of the logical xor operator
$\texttt{ift}(x == 0, 5, 10)$	the value is 5 or 10 according to the value of x

Table 1.1: A few examples of expressions denoting `intension` constraints.

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

Relational 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 1.2: 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`. Note that we use the operators |, & and ^ for logically combining (sub-)expressions. We can't use the Python operators `and`, `or` and `not` (because they cannot be redefined).

Functions	
`abs`()	absolute value of the argument
`min`()	minimum value of 2 or more arguments
`max`()	maximum value of 2 or more arguments
`dist`()	distance between the 2 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 1.3: Functions that can be used to build expressions (predicates) of `intension` constraints.

Remember that in this first chapter, XCSP^3 files are given for well understanding what models represent (and how models are compiled), but if you think that it does not make things clearer for you, you can decide to ignore them. As a user working with the PyCSP^3 library and some constraint solvers, you may never need to look at these intermediate XCSP^3 files (although, by experience, it may be helpful in identifying some mistakes in models and some bugs in solvers).

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 (version) in a file called 'Riddle2.py':

PyCSP^3 Model 2

```python
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 executing:

```
python3 Riddle2.py
```

is:

```xml
<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>
    <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))

...  # the rest of the code is similar to the previous model
```

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

```python
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 `sum` constraint. 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

```python
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:

```xml
<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 1.2 presents the compatibility graph of a small constraint network P such that:

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

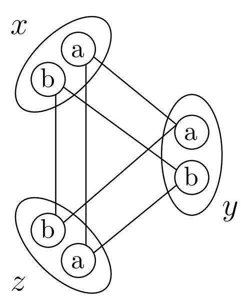


Figure 1.2: The compatibility graph of a small constraint network.

Here, the constraints directly indicate what is authorized; we call such constraints `extension` constraints (or table constraints). For example, we know that we can satisfy the binary constraint involving the variables x and y by assigning both variables with either value a or value b. 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.

The PyCSP^3 model for our problem, in a file called 'Pic.py', is:

PyCSP^3 Model 8

```
from pycsp3 import *

a, b = "a", "b"  # two symbols

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 `extension` constraints. 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), as in our example, 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)]
```

but personally, we prefer to stay closer to pure mathematical forms (but for efficiency reasons, we may use lists for huge tables).

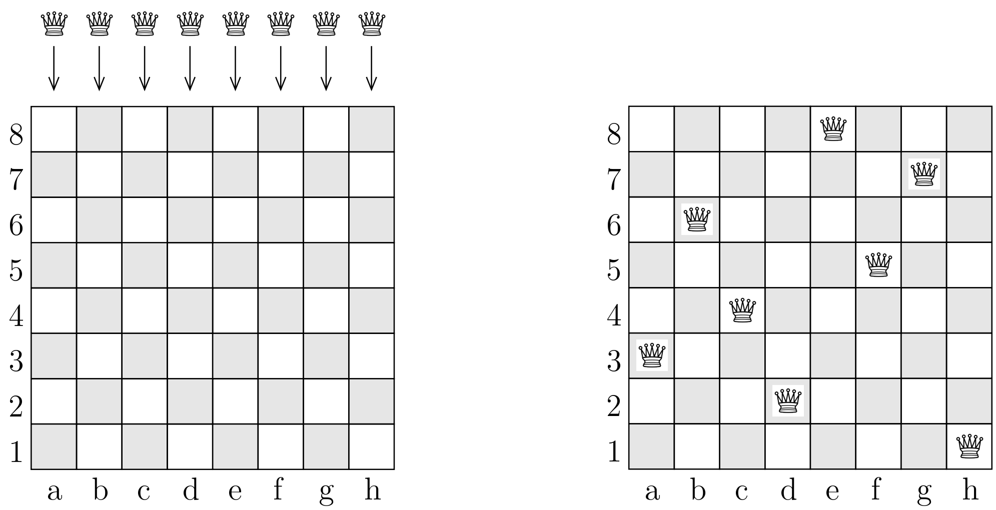
1.2 Academic Problems

Contrary to single problems, *academic* problems require the introduction of some elementary pieces of data from the user: a fixed number of integers (and/ or strings).

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 1.3.

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 specific value of n. We can try to solve the 8-queens instance, the 10-queens instance, and even the 1000-queens instance.



(a) Puzzle (b) Solution

Figure 1.3: Putting 8 queens on a chessboard

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 means of the predefined PyCSP^3 variable called `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 $(i+1)$th row of the board, then we can simply post the following constraints:

$$q_i \neq q_j \wedge |q_i - q_j| \neq |i - j|, \forall i, j : 0 \leq i < j < n$$

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

PyCSP^3 Model 10

```
from pycsp3 import *

n = data

# 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, j in combinations(range(n), 2)
)
```

Note how the parameter n is given by the value of the predefined PyCSP^3 variable `data`. This is because there is only one parameter here; later, we shall see that for more than one parameter, data is given under the form of a tuple. The intension constraints are given by a comprehension list (actually a generator, since brackets are omitted here although we could have inserted them). There is a constraint for any pair (i,j) such that $0 \leq i < j < n$. Here, for iterating over such pairs, we use the (automatically imported) function combinations from package itertools. Instead, we could have written :

```
for i in range(n) for j in range(i + 1, n)
```

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
```

In our situation where only one integer is needed (and more generally, for any academic problem), it is a little bit of overkill to use JSON files.

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 (you can try to make the mathematical proof) that it suffices to post three constraints as in the following model:

PyCSP^3 Model 11

```python
from pycsp3 import *

n = data

# 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:

```xml
<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 2 *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

```python
from pycsp3 import *

n = data

# 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, j in combinations(range(n), 2)
    )
```

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 n rows and m 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 1.4: Coloring Boards. (image by Ylanite Koppens on Pixabay)

This time, we then need two integer parameters n and m. These values will be given by the predefined PyCSP^3 variable `data` that is expected to be a tuple (if data are correctly given at compile time, of course). 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

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

```
satisfy(
   # at least 2 corners of different colors for any rectangle inside the board
   NValues(x[i1][j1], x[i1][j2], x[i2][j1], x[i2][j2]) > 1
      for i1, i2 in combinations(range(n), 2)
      for j1, j2 in combinations(range(m), 2)
)

minimize(
   # minimizing the greatest used color index (and so, the number of colors)
   Maximum(x)
)
```

The user is expected to give two integer values, automatically put in `data` under the form of a tuple. This is why we have the possibility of using tuple unpacking in our model. Of course, this is equivalent to write:

```
n, m = data[0], data[1]
```

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 then need to post several `notAllEqual` constraints. Actually, this constraint is a special case of the `nValues` constraint: 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)`

As we shall see later, it is even possible to build still more general (arithmetic) expressions involving functions `Sum`(), `Minimum`(), etc.

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

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

With some operating systems (shells), you may need to espace brackets, which gives:

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

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 note="at least 2 corners of different colors for any rectangle">
      <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>
      ...  // ellipsis
      <args> x[1][1] x[1][2] x[2][2] x[2][3] </args>
    </group>
  </constraints>
  <objectives>
    <minimize type="maximum"> x[][] </minimize>
  </objectives>
</instance>
```

Of course, because tuple unpacking is used for data in our model, the order is important: the first value is for n and the second one for m. If ever we use a JSON file for the data, it is also important to have n before m:

```
{
  "n": 3,
  "m": 4
}
```

However, you can relax this requirement by avoiding tuple unpacking for data, and instead write in the model something like:

```
n, m = data.n, data.m
```

It means that `data` is now expected to be a named tuple (and not simply a classical tuple). To benefit from named tuples, you have to either indicate names when specifying data, as for example, in:

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

or use a JSON file (whatever is the order of the fields of the root object in the file).

This being said, we prefer personnally to use tuple unpacking for data because it is more concise.

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 white-space) between parentheses. The model is now:

PyCSP^3 Model 14

```
from pycsp3 import *

n, m = data

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

```
satisfy(
   # at least 2 corners of different colors for any rectangle inside the board
   [NValues(x[i1][j1], x[i1][j2], x[i2][j1], x[i2][j2]) > 1
      for i1, i2 in combinations(range(n), 2)
      for j1, j2 in combinations(range(m), 2)],

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

minimize(
   # minimizing the greatest used color index (and so, the number of colors)
   Maximum(x)
)
```

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

```
<lex class="symmetry-breaking">
  <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, which 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 mathematically 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 for improving the filtering process of the search space while 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

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

```python
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 `cardinality` constraint 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 required named parameter called `occurrences` is given as value 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:

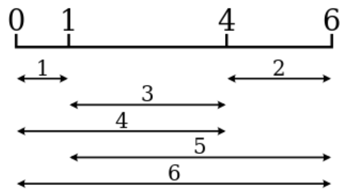
```
python3 MagicSequence.py -data=6
```

we obtain the following XCSP^3 instance:

```xml
<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

# 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, j in combinations(range(n), 2)
)

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

Another 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

# 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, j in combinations(range(n), 2)]
)

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

```python
from pycsp3 import *

n = data

# 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, j in combinations(range(n), 2)],

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

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

For $n = 4$, we obtain:

```xml
<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>
```

Technically, the undefined variables of the array y in the PyCSP^3 model are not identified as such in the XCSP^3 instance (see the element `<array>` for y). However, although not explicitly identified as undefined, they can be discarded by solvers because they are involved nowhere (constraint or objective); see how the constraint `<allDifferent>` only involves the variables in the upper half of the two-dimensional array y.

1.3 Structured Problems

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

1.3.1 Sudoku

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

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

An illustration is given by Figure 1.5.

	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 1.5: Solving a Sudoku Grid

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  # 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]],

   # imposing clues  tag(clues)
   [x[i][j] == clues[i][j] for i in range(9) for j in range(9)
     if clues and 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 as not all clues are shown):

```
<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
```

or

```
python3 Sudoku.py -data=null
```

or even

```
python3 Sudoku.py
```

For these three last commands, the value `None` is set to the predefined PyCSP^3 variable `data`.

1.3.2 Warehouse Location

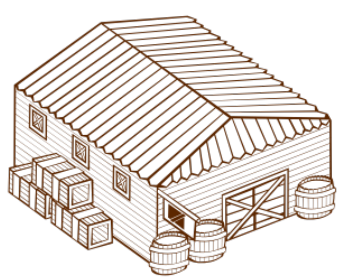


Figure 1.6: A Warehouse. (image from freesvg.org)

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]
  ]
}
```

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

PyCSP^3 Model 20

```
from pycsp3 import *

wcost, capacities, costs = data  # wcost is the fixed cost when opening a warehouse
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: 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) * wcost
)
```

Concerning data, the root object in the JSON file is expected to have three fields. We then expect to get a named tuple of size 3 that can be unpacked. An alternative is to write something like:

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

Here, we associate a specific domain with each variable of the array c by means of a lambda function. Note that it is possible to give a list, `costs[i]`, instead of a set, `set(costs[i])`, as the list will be automatically converted to a set. For dealing with warehouse capacities, we use the `count` constraint: 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 `element` constraint: 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 `element` constraint 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>
```

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 1.7.

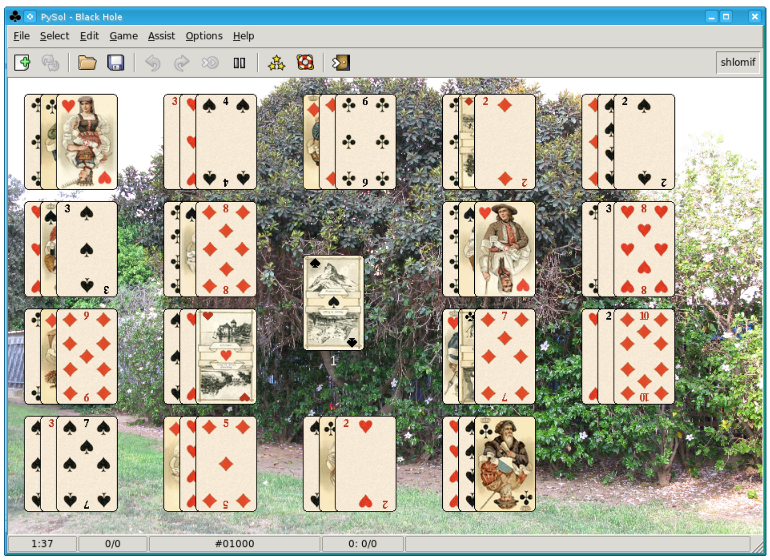


Figure 1.7: Blackhole. (image by Shlomif on commons.wikimedia.org)

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-4.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
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(
   # linking variables of x and y
   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 (we have $x[i] = j \Leftrightarrow y[j] = i$), 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`, introduced in Section 3.21. 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 note="linking variables of x and y">
      <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>
```

Here, the main interest of using `slide` is that the generated XCSP^3 file is made compacter (while emphasizing the sliding structure). However, in our illustration, because the sliding form is not circular and because two successive constraints only share one variable, any solver reasoning individually with the sliding constraints will reach the same efficiency (i.e., will reach the same level of filtering of the search space) as reasoning with the meta-constraint.

If you are worried by using the PyCSP^3 function `Slide()` in the model, you can let the model as it was given initially, and use the option `-recognizeSlides` as in the following command:

```
python3 Blackhole.py -data=blackhole-4.json -recognizeSlides
```

1.3.4 Rack Configuration

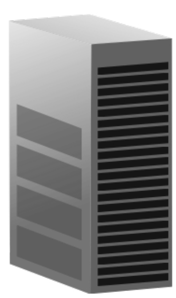


Figure 1.8: A Rack. (image from freesvg.org)

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
models.append([0, 0, 0])  # we add first a dummy model (0,0,0)
powers, sizes, costs = zip(*models)
cardPowers, cardDemands = zip(*cardTypes)
nModels, nTypes = len(models), len(cardTypes)

table = {(i, powers[i], sizes[i], costs[i]) for i in range(nModels)}

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

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

# s[i] is the size (number of connectors) of the model used for the ith rack
s = VarArray(size=nRacks, dom=sizes)

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

# 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))

satisfy(
   # linking rack models with powers, sizes and costs
   [(m[i], p[i], s[i], c[i]) in table 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), imply(m[0] == m[1], nc[0][0] >= nc[1][0])]
)

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

From data, we build first some auxiliary lists that will be useful for writing our model. Note that using the Python function `zip()` is simpler than writing for example:

```
cardPowers, cardDemands = [row[0] for row in cardTypes], [row[1] for row in cardTypes]
```

After declaring five arrays of variables, a quaternary table constraint is first posted. See how it is easy to link variables of 4 arrays with a simple table. 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.

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 variable `data` is used in the model. In our case, with this new representation, we only need to replace:

```
models.append([0, 0, 0])  # we add first a dummy model (0,0,0)
```

with:

```
models.append(models[0].__class__(0, 0, 0))  # we add first a dummy model (0,0,0)
```

Again we add a dummy rack model to those defined in the JSON file. To do that, and in order to avoid breaking the homogeneity of the data, we get the class of the used named tuples to build and add a new one. As any JSON object is automatically converted to a named tuple, we still have the possibility to use the function `zip()` in our model.

Chapter 2

Data, Variables and Objectives

In this chapter, we give some additional details and illustrations about data, variables and objectives, although many examples can already be found in the other chapters.

2.1 Specifying Data

Except for "single" problems, each problem usually represents a large (often, infinite) family of cases, called instances, that one may want to solve. All these instances are uniquely identified by some specific data.

First, recall that the command to be run for generating an XCSP^3 instance (file), given a model and some data is:

```
python3 <model_file> -data=<data_values>
```

where <model_file> (is a Python file that) represents a PyCSP^3 model, and <data_values> represents some specific data. In our context, an *elementary* value is a value of one of these built-in data types: integer ('int'), real ('float'), string ('str') and boolean ('bool'). Specific data can be given as:

1. a single elementary value, as in `-data=5`
2. a list of elementary values, between square (or round) brackets[1] and with comma used as a separator, as in `-data=[9,0,0,3,9]`
3. a list of named elementary values, between square (or round) brackets and with comma used as a separator, as in `-data=[v=9,b=0,r=0,k=3,l=9]`
4. a JSON file, as in `-data=Bibd-9-3-9.json`
5. a file in any arbitrary format (e.g., a text file) while providing with the option `-dataparser` some Python code to load it, as in `-data=puzzle.txt -dataparser=ParserPuzzle.py`

Then, **data can be directly used in** PyCSP^3 **models by means of a predefined variable called** `data`. The value of the predefined PyCSP^3 variable `data` is set as follows:

1. if the option `-data` is not specified, or if it is specified as `-data=null` or `-data=None`, then the value of `data` is `None`. See, for example, Section 1.3.1.
2. if a single elementary value is given (possibly, between brackets), then the value of `data` is directly this value. See, for example, Section 1.2.4.

[1] According to the operating system, one might need to escape brackets.

3. if a JSON file containing a root object with only one field is given, then the value of `data` is directly this value. See, for example, Section 1.3.1.
4. if a list of (at least two) elementary values is given, then the value of `data` is a tuple containing those values in sequence. See, for example, Section 1.2.2.
5. if a list of (at least two) named elementary values is given, then the value of `data` is a named tuple. See, for example, Section 1.2.2.
6. if a JSON file containing a root object with at least two fields is given, then the value of `data` is a named tuple. Actually, any encountered JSON object in the file is (recursively) converted into a named tuple. See, for example, Section 1.3.2 and Section 1.3.4.

Although various cases have already been illustrated in Chapter 1, we introduce below a few additional examples.

All-Interval Series. Given the twelve standard pitch-classes (c, c#, d, ...), represented by numbers $0, 1, \ldots, 11$, find a series in which each pitch-class occurs exactly once and in which the musical intervals between neighboring notes cover the full set of intervals from the minor second (1 semitone) to the major seventh (11 semitones). That is, for each of the intervals, there is a pair of neighboring pitch-classes in the series, between which this interval appears.

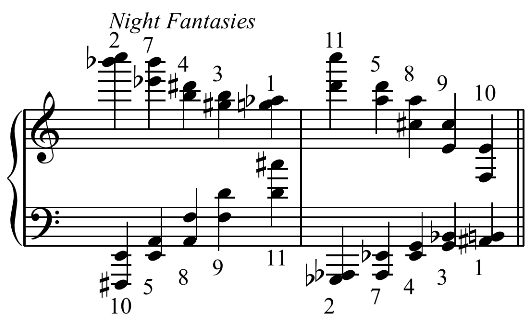


Figure 2.1: Elliott Carter often bases his all-interval sets on the list generated by Bauer-Mendelberg and Ferentz and uses them as a "tonic" sonority (image from commons.wikimedia.org)

The problem of finding such a series can be easily formulated as an instance of a more general arithmetic problem. Given a positive integer n, find a sequence $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$, such that:

1. x is a permutation of $\{0, 1, ..., n-1\}$;
2. the interval sequence $y = \langle |x_1 - x_0|, |x_2 - x_1|, ...|x_{n-1} - x_{n-2}| \rangle$ is a permutation of $\{1, 2, ..., n-1\}$.

A sequence satisfying these conditions is called an all-interval series of order n; the problem of finding such a series is the all-interval series problem of order n. For example, for $n = 8$, a solution is:

```
1 7 0 5 4 2 6 3
```

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

PyCSP^3 Model 23

```
from pycsp3 import *

n = data

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

satisfy(
   # notes must occur once, and so form a permutation
   AllDifferent(x),

   # intervals between neighbouring notes must form a permutation
   AllDifferent(abs(x[i] - x[i + 1]) for i in range(n - 1)),
)
```

Here, the required data is a single integer value. So, to generate the XCSP^3 instance of `AllInterval` for order 12, we just execute:

```
python3 AllInterval.py -data=12
```

Balanced Incomplete Block Designs. From CSPLib: "Balanced Incomplete Block Design (BIBD) generation is a standard combinatorial problem from design theory, originally used in the design of statistical experiments but since finding other applications such as cryptography. It is a special case of Block Design, which also includes Latin Square problems. BIBD generation is described in most standard textbooks on combinatorics. A BIBD is defined as an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks. Another way of defining a BIBD is in terms of its incidence matrix, which is a v by b binary matrix with exactly r ones per row, k ones per column, and with a scalar product of λ between any pair of distinct rows. A BIBD is therefore specified by its parameters (v, b, r, k, λ)."

An example of a solution for $(7, 7, 3, 3, 1)$ is:

$$\begin{matrix} 0&1&1&0&0&1&0 \\ 1&0&1&0&1&0&0 \\ 0&0&1&1&0&0&1 \\ 1&1&0&0&0&0&1 \\ 0&0&0&0&1&1&1 \\ 1&0&0&1&0&1&0 \\ 0&1&0&1&1&0&0 \end{matrix}$$

Hence, we need five integers v, b, r, k, l (for λ) for specifying a unique instance; possibly, b and r can be set to 0, so that these values are automatically computed according to a classical BIBD template. A PyCSP^3 model of this problem is given by the following file 'Bibd.py':

PyCSP^3 Model 24

```
from pycsp3 import *

v, b, r, k, l = data
b = (l * v * (v - 1)) // (k * (k - 1)) if b == 0 else b
r = (l * (v - 1)) // (k - 1) if r == 0 else r

# x[i][j] is the value of the matrix at row i and column j
x = VarArray(size=[v, b], dom={0, 1})
```

```
satisfy(
   # constraints on rows
   [Sum(row) == r for row in x],

   # constraints on columns
   [Sum(col) == k for col in columns(x)],

   # scalar constraints with respect to lambda
   [row1 * row2 == l for row1, row2 in combinations(x, 2)]
)
```

To generate an XCSP^3 instance (file), we can for example execute:

```
python3 Bibd.py -data=[9,0,0,3,9]
```

As mentioned earlier, with some command interpreters (shells), you may have to escape the characters '[' and ']', which gives:

```
python3 Bibd.py -data=\[9,0,0,3,9\]
```

You can also use round brackets instead of square brackets:

```
python3 Bibd.py -data=(9,0,0,3,9)
```

If it causes some problem with the command interpreter (shell), you have to escape the characters '(' and ')', which gives:

```
python3 Bibd.py -data=\(9,0,0,3,9\)
```

Suppose that you would prefer to have a JSON file for storing these data values. You can execute:

```
python3 Bibd.py -data=[9,0,0,3,9] -datexport
```

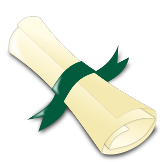
You then obtain the following JSON file 'Bibd-9-0-0-3-9.json'

```
{
  "v":9,
  "b":0,
  "r":0,
  "k":3,
  "l":9
}
```

And now, to generate the same XCSP^3 instance (file) as above, you can execute:

```
python3 Bibd.py -data=Bibd-9-0-0-3-9.json
```

Balanced Academic Curriculum Problem (BACP). From CSPLib: "The goal of BACP is to design a balanced academic curriculum by assigning periods to courses in a way that the academic load of each period is balanced, i.e., as similar as possible. An academic curriculum is defined by a set of courses and a set of prerequisite relationships among them. Courses must be assigned within a maximum number of academic periods. Each course is associated to a number of credits or units that represent the academic effort required to successfully follow it.



The curriculum must obey the following regulations:

- minimum academic load: a minimum number of academic credits per period is required to consider a student as full time
- maximum academic load: a maximum number of academic credits per period is allowed in order to avoid overload
- minimum number of courses: a minimum number of courses per period is required to consider a student as full time
- maximum number of courses: a maximum number of courses per period is allowed in order to avoid overload

The goal is to assign a period to every course in a way that the minimum and maximum academic load for each period, the minimum and maximum number of courses for each period, and the prerequisite relationships are satisfied. An optimal balanced curriculum minimizes the maximum academic load for all periods."

When analyzing this problem, we identify its parameters as being the number of periods (an integer), the minimum and the maximum number of credits (two integers), the minimum and the maximum number of courses (two integers), the credits for each course (a one-dimensional array of integers) and the prerequisites (a two-dimensional array of integers, with each row indicating a prerequisite). An example of data is given by the following JSON file:

```
{
  "nPeriods": 4,
  "minCredits": 2,
  "maxCredits": 5,
  "minCourses": 2,
  "maxCourses": 3,
  "credits": [2,3,1,3,2,3,3,2,1],
  "prequisites": [[2,0],[4,1],[5,2],[6,4]]
}
```

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

PyCSP^3 Model 25

```
from pycsp3 import *

nPeriods, minCredits, maxCredits, minCourses, maxCourses, credits, prereq = data
nCourses = len(credits)

# s[c] is the period (schedule) for course c
s = VarArray(size=nCourses, dom=range(nPeriods))

# co[p] is the number of courses at period p
co = VarArray(size=nPeriods, dom=range(minCourses, maxCourses + 1))

# cr[p] is the number of credits at period p
cr = VarArray(size=nPeriods, dom=range(minCredits, maxCredits + 1))

# cp[c][p] is 0 if the course c is not planned at period p,
#              the number of credits for c otherwise
cp = VarArray(size=[nCourses, nPeriods], dom=lambda c, p: {0, credits[c]})

def table(c):
   return {(0,) * p + (credits[c],) + (0,) * (nPeriods - p - 1) + (p,)
     for p in range(nPeriods)}
```

```
satisfy(
   # channeling between arrays cp and s
   [(*cp[c], s[c]) in table(c) for c in range(nCourses)],

   # counting the number of courses in each period
   [Count(s, value=p) == co[p] for p in range(nPeriods)],

   # counting the number of credits in each period
   [Sum(cp[:, p]) == cr[p] for p in range(nPeriods)]

   # handling prerequisites
   s[c1] < s[c2] for (c1, c2) in prereq
)

minimize(
   # minimizing the maximum number of credits in periods
   Maximum(cr)
)
```

Because tuple unpacking is used, it is important to note that the fields of the root object in the JSON file must be given in this exact order. If it is not the case, as for example:

```
{
  "nPeriods": 4,
  "prequisites": [[2,0],[4,1],[5,2],[6,4]],
  "minCredits": 2,
  "maxCredits": 5,
  "credits": [2,3,1,3,2,3,3,2,1],
  "minCourses": 2,
  "maxCourses": 3
}
```

there will be a problem when unpacking data. If you wish a safer model (because, for example, you have no guarantee about the way the data are generated), you must specifically refer to the fields of the named tuple instead:

```
from pycsp3 import *

nPeriods = data.nPeriods
minCredits, maxCredits = data.minCredits, data.maxCredits
minCourses, maxCourses = data.minCourses, data.maxCourses
credits, prereq = data.credits, data.prerequisites
nCourses = len(credits)
```

Now, let us suppose that you would like to use the data from this MiniZinc file 'bacp-data.mzn':

```
include "curriculum.mzn.model";
n_courses = 9;
n_periods = 4;
load_per_period_lb = 2;
load_per_period_ub = 5;
courses_per_period_lb = 2;
courses_per_period_ub = 3;
course_load = [2, 3, 1, 3, 2, 3, 3, 2,1, ];
constraint prerequisite(2, 0);
constraint prerequisite(4, 1);
constraint prerequisite(5, 2);
constraint prerequisite(6, 4);
```

We need to write a piece of code in Python for building the variable `data` that will used in our model. After importing everything (*) from `pycsp3.problems.data.parsing`, we can use some PyCSP^3 functions such as `next_line()`, `number_in()`, `remaining_lines()`, ... Here, we also use the classical function `split()` of module `re` to parse information concerning prerequisites. Note that you have to add relevant fields to the predefined dictionary[2] `data`, as in the following file 'Bacp_ParserZ.py':

```
from pycsp3.problems.data.parsing import *

nCourses = number_in(next_line())
data["nPeriods"] = number_in(next_line())
data["minCredits"] = number_in(next_line())
data["maxCredits"] = number_in(next_line())
data["minCourses"] = number_in(next_line())
data["maxCourses"] = number_in(next_line())
data["credits"] = numbers_in(next_line())
data["prerequisites"] = [[int(v) - 1
    for v in re.split(r'constraint prerequisite\(|,|\);', line) if len(v) > 0]
      for line in remaining_lines(skip_curr=True)]
```

To generate the XCSP^3 instance (file), you have to execute:

```
python3 Bacp.py -data=bacp.mzn -dataparser=Bacp_ParserZ.py
```

If you want the same data put in a JSON file, execute:

```
python3 Bacp.py -data=bacp-data.mzn -dataparser=Bacp_ParserZ.py -dataexport
```

You obtain a file called 'bacp-data.json' equivalent to the one introduced earlier. If you want to specify the name of the output JSON file, give it as a value to the option `-dataexport`, as e.g., in:

```
python3 Bacp.py -data=bacp-data.mzn -dataparser=Bacp_ParserZ.py -dataexport=instance0
```

The generated JSON file is then called 'instance0.json'.

Special Rules when Loading JSON Files. The rules that are used when loading a JSON file in order to set the value of the PyCSP^3 predefined variable `data` are as follows.

1. For any field f of the root object in the JSON file, we obtain a field `f` in the generated named tuple `data` such that:
   - if `f` is a JSON list (or recursively, a list of lists) containing only integers, the type of `data.f` is 'pycsp3.tools.curser.ListInt' instead of 'list'; 'ListInt' being a subclass of 'list'. The main interest is that `data.f` can be directly used as a vector for the global constraint `element`. See Mario Problem, page 91, for an illustration.
   - if `f` is an object, `data.f` is a named tuple with the same fields as `f`. See Rack Configuration Problem in Section 1.3.4 for an illustration.
2. The rules above applies recursively.

Special Rule when Building Arrays of Variables. When we define a list (array) x of variables with `VarArray()`, the type of x is 'pycsp3.tools.curser.ListVar' instead of 'list'. The main interest is that x can be directly used as a vector for the global constraint `element`.

Special Values `null` and `None`. When the value `null` occurs in a JSON file, it becomes `None` in PyCSP^3 after loading the data file. An illustration is given at the end of Section 1.3.1.

[2] At this stage, `data` is a dictionary. Later, it will be automatically converted to a named tuple.

2.2 Declaring Variables

2.2.1 Stand-alone Variables

Stand-alone variables can be declared by means of the PyCSP^3 function `Var()`. 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(set(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))   # or equivalently, w = Var(dom=set(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 generators and comprehension sets. 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 still equivalently:

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

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

2.2.2 Arrays of Variables

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. The named parameter `dom` indicates the domain of each variable in the array.

The signature of the function `VarArray()` is:

```
def VarArray(*, size, dom):
```

An illustration is given by:

```
x = VarArray(size=10, dom={0, 1})
y = VarArray(size=[5, 20], dom=range(10))
z = VarArray(size=[4, 3, 4], dom={1, 5, 10, 20})
```

We have:

- x, a one-dimensional array of 10 variables with domain $\{0, 1\}$
- y, a two-dimensional array of 5×20 variables with domain $\{0, 1, \dots, 9\}$
- z, a three-dimensional array of $4 \times 3 \times 4$ variables with domain $\{1, 5, 10, 20\}$

Indexing starts at 0. For example, $x[2]$ is the third variable of x, and $y[1]$ is the second row of y. Technically, variable arrays are objects that are instances of `ListVar`, a subclass of `list`; additional functionalities of such objects are useful, for example, when posting the `element` constraint.

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 the domain of all variables of the first column of y is `range(5)` instead of `range(10)`. We can write:

```
def domain_y(i,j):
   return range(5) if j == 0 else range(10)

y = VarArray(size=[5, 20], dom=domain_y)
```

We can also use a lambda function:

```
y = VarArray(size=[5, 20], dom=lambda i,j: range(5) if j == 0 else range(10))
```

Sometimes, not all variables in an array are relevant. For example, you may only want to use the variables in the lower part of a two-dimensional array (matrix). In that case, the value `None` must be used. An illustration is given below:

Golomb Ruler. This problem was introduced in Section 1.2.4. Here is a snippet of the PyCSP^3 model:

```
# 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)
```

In the array y, the upper part (above the main downward diagonal) only contains `None`. For example, $y[1][0]$ is equal to `None`. This is taken into consideration when the XCSP^3 file is generated by compilation.

Sometimes, one may want to be able to refer to variables in arrays in an individual manner. It suffices to use facilities offered by Python, as shown in the following model.

Allergy. Four friends (two women named Debra and Janet, and two men named Hugh and Rick) found that each of them is allergic to something different: eggs, mold, nuts and ragweed. We would like to match each one's surname (Baxter, Lemon, Malone and Fleet) with his or her allergy. We know that:

- Rick isn't allergic to mold
- Baxter is allergic to eggs
- Hugh isn't surnamed Lemon or Fleet
- Debra is allergic to ragweed

◦ Janet (who isn't Lemon) isn't allergic to eggs or mold

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

PyCSP^3 Model 26

```
from pycsp3 import *

Debra, Janet, Hugh, Rick = friends = ["Debra", "Janet", "Hugh", "Rick"]

# foods[i] is the friend allergic to the ith food
eggs, mold, nuts, ragweed = foods = VarArray(size=4, dom=friends)

# surnames[i] is the friend with the ith surname
baxter, lemon, malone, fleet = surnames = VarArray(size=4, dom=friends)

satisfy(
    AllDifferent(foods),
    AllDifferent(surnames),

    mold != Rick,
    eggs == baxter,
    lemon != Hugh,
    fleet != Hugh,
    ragweed == Debra,
    lemon != Janet,
    eggs != Janet,
    mold != Janet
)
```

Note how we define an array of variables, and unpack its elements. This way, we can reason with either the array or individual variables. Any comment put in the line preceding the declaration of a variable (or variable array) is automatically inserted in the XCSP^3 file, except for cases where individual variables and arrays are declared on the same line, as in the model above.

2.3 Specifying Objectives

For specifying an objective to optimize, you must call one of the two functions:

```
def minimize(term):
```

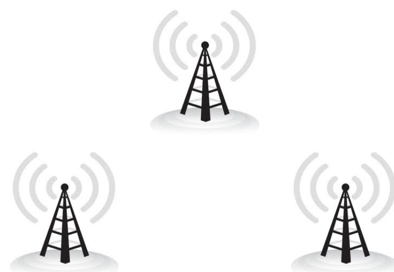
```
def maximize(term):
```

The argument `term` can be:

- a variable, as in `minimize(v)`
- an expression, as in `minimize(v + w * w)`
- a sum, as in `minimize(Sum(x))`
- a dot product, as in `minimize([u,v,w] * [3, 2, 5])`
- a generator, as in `minimize(Sum((x[i] > 1) * c[i] for i in range(n)))`
- a minimum, as in `minimize(Minimum(x))`
- a maximum, as in `minimize(Maximum(x))`
- a number of distinct values, as in `minimize(NValues(x))`
- ...

An illustration is given by the three different variants of the following problem.

RLFAP. From Cabon et al. [7]: "When radio communication links are assigned the same or closely related frequencies, there is a potential for interference. Consider a radio communication network, defined by a set of radio links. The radio link frequency assignment problem [7] is to assign, from limited spectral resources, a frequency to each of these links in such a way that all the links may operate together without noticeable interference. Moreover, the assignment has to comply to certain regulations and physical constraints of the transmitters. Among all such assignments, one will naturally prefer those which make good use of the available spectrum, trying to save the spectral resources for a later extension of the network.



Formal Definition: we are given a set X of unidirectional radio links. For each link $i \in X$, a frequency f_i has to be chosen from a finite set D_i of frequencies available for the transmitter which yield unary constraints of type:

$$f_i \in D_i \tag{2.1}$$

Depending on the type of the problem (bulk or updating problem), some links may already have a pre-assigned frequency which define unary constraints of the type

$$f_i = p_i \tag{2.2}$$

Binary constraints are defied on pairs of links $\{i, k\}$. These constraints may be either of type:

$$|f_i - f_j| > d_{ij} \tag{2.3}$$

or of type:

$$|f_i - f_j| = d_{ij} \tag{2.4}$$

Depending on the instance considered, some of the constraints may actually be soft constraints which may be violated at some cost. A mobility cost m is defied for changing pre-assigned values, defined by constraints of type 2.2 and an interference cost c is defined for violation of soft constraints of type 2.3. Constraints of type 2.1 and 2.4 are always hard. The complete set of constraints C is therefore partitioned in a set H of hard constraints and a set S of soft constraints. Several variants can be defined:

1. Minimum span (SPAN): if all the constraints in C can be satisfied together, one can try to minimize the largest frequency used in the assignment.
2. Minimum cardinality (CARD): if all the constraints in C can be satisfied together, one can try to minimize the number of different frequencies used in the assignment.
3. Maximum Feasibility (MAX): if all the constraints in C cannot be satisfied simultaneously, one should try to find an assignment that satisfies all constraints in H and that minimizes the sum of all the violation costs (interference cost and mobility cost) for constraints in S.

"

As an illustration of data specifying an instance of this problem, we have:

```
{
  "domains": [
    [16, 30, 44, 58, 72, 86, 100, 114, 128, 142, 156, 254, 268, ...],
    [30, 58, 86, 114, 142, 268, 296, 324, 352, 380, 414, 442, 470, ...],
    ...
  ],
  "vars": [
    {"domain": 0, "value": null, "mobility": null },
    {"domain": 1, "value": 58, "mobility": 0 },
    ...
  ],
  "ctrs":[
    {"x": 13, "y": 14, "operator": ">", "limit": 238, "weight": 0 },
    {"x": 13, "y": 16, "operator": "=", "limit": 186, "weight": 1 },
    ...
  ],
  "mobilityCosts": [0, 0, 0, 0, 0],
  "interferenceCosts": [0, 1000, 100, 10, 1]
}
```


The fields `mobility` and `weight` are indexes for getting the actual cost in the two arrays `mobilityCosts` and `interferenceCosts`. For more details, we refer the reader to [7].

PyCSP^3 Model 27

```
from pycsp3 import *

domains, variables, constraints, mobilityCosts, interferenceCosts = data
n = len(variables)

# f[i] is the frequency of the ith radio link
f = VarArray(size=n, dom=lambda i: domains[variables[i].domain])

satisfy(
   # managing pre-assigned frequencies
   [f[i] == v for i, (_, v, mob) in enumerate(variables)
     if v and not (variant("max") and mob)],

   # hard constraints on radio-links
   [expr(op, abs(f[i] - f[j]), k) for (i, j, op, k, wgt) in constraints
     if not (variant("max") and wgt)]
)

if variant("span"):
   minimize(
      # minimizing the largest frequency
      Maximum(f)
   )
elif variant("card"):
   minimize(
      # minimizing the number of used frequencies
      NValues(f)
   )
elif variant("max"):
   minimize(
      # minimizing the sum of violation costs
      Sum(ift(f[i] == v, 0, mobilityCosts[mob])
        for (i, v, mob) in variables if v and mob)
      + Sum(ift(expr(op, abs(f[i] - f[j]), k), 0, interferenceCosts[wgt])
        for (i, j, op, k, wgt) in constraints if wgt)
   )
```

Constraints of types 2.2 and 2.3 are considered to be hard when the variant is not "max" or the index (for mobility/interference cost) is not 0. Note that we use the PyCSP^3 function `expr()` to post the binary constraint on pairs of links; the first parameter is a string denoting an operator that can be chosen in $\{<, <=, >=, >, =, ==, !=, lt, le, ge, gt, eq, ne, \dots\}$. In our context, the code

```
expr(op, abs(f[i] - f[j]), k)
```

is equivalent to:

```
abs(f[i] - f[j]) == k if op == "=" else abs(f[i] - f[j]) > k
```

Concerning the objective, we have three kinds of minimization. Note how we can combine several partial computations (here, sums), when dealing with the variant "max". Remember that the PyCSP^3 ternary function `ift()` (if-then-else) returns either the second parameter or the third parameter according to the fact the first parameter evaluates to `True` or `False`.

Chapter 3

Twenty Popular Constraints

In this chapter, we introduce twenty popular constraints, those from XCSP^3-core that are recognized by many constraint solvers. Figure 3.1 shows their classification.

Semantics. Concerning the semantics of constraints, 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 `Sum(x) > 10`, `Count(x, value = 1) in range(10)`, `NValues(x) in {2, 4, 6}` and `Minimum(x) == y`. Of course, we can also write `10 < Sum(x)` and `y == 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 Table1.2 and Table 1.3. In Table 1.1, 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 maybe more relevant) to write it differently.

- **Constraints over Integer Variables**
  - Generic Constraints
    - Constraint `intension`
    - Constraint `extension`
  - Language-based Constraints
    - Constraint `regular`
    - Constraint `mdd`
  - Comparison-based Constraints
    - Constraints `allDifferent`, `allDifferentList`, `allEqual`
    - Constraints `increasing`, `decreasing`
    - Constraints `lexIncreasing`, `lexDecreasing`
  - Counting Constraints
    - Constraint `sum`
    - Constraint `count`
    - Constraint `nValues`
    - Constraint `cardinality`
  - Connection Constraints
    - Constraint `maximum`
    - Constraint `minimum`
    - Constraint `element`
    - Constraint `channel`
  - Packing and Scheduling Constraints
    - Constraint `noOverlap`
    - Constraint `cumulative`
  - Other Constraints
    - Constraint `circuit`
    - Constraint `slide`

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`
$P(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{r-1}) = \mathit{true}$ (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.

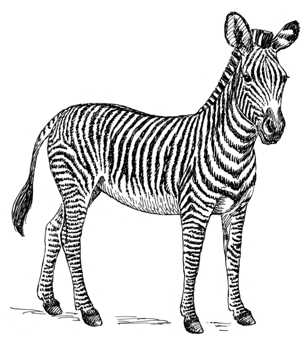


Figure 3.2: In which house lives the zebra? (image from /commons.wikimedia.org)

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 28

```
from pycsp3 import *

houses = range(5)  # each house has a number from 0 (left) to 4 (right)

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

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

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

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

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

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, notably, with the possibility of using the operator `in`. Note how we define arrays of variables and unpack them so as to simplify the task of posting constraints. For example, `colors` is an array of 5 variables, the first one `colors[0]` being given `yellow` as alias, the second one `colors[1]` being given `green` as alias, and so on.

Important. Note that we use the operators |, & and ^ for logically combining (sub-)expressions. We can't use the Python operators `and`, `or` and `not` (because they cannot be redefined).

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 (a set of tuples of values) ⟨`table`⟩ as follows:

⟨`scope`⟩ $\in$ ⟨`table`⟩

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

⟨`scope`⟩ $\notin$ ⟨`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.

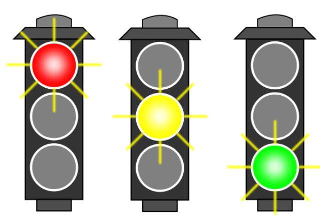


Figure 3.3: How to adjust traffic lights? (image from freesvg.org)

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 modeled by quaternary constraints on (v_i, p_i, v_j, p_j) for $1 \leq i \leq 4, j = (1+i) \,\mathtt{mod}\, 4$ which allow just the tuples $\{(r, r, g, g), (ry, r, y, r), (g, g, r, r), (y, r, ry, r)\}$."

PyCSP^3 Model 29

```
from pycsp3 import *

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

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

# 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})

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 [26] 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 $i \leq j, d_{i,j} = d_{j,i} = \min(j-i, i-j+n)$."

Figure 3.4: Traveling Tournament (image from freesvg.org)

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 30

```
from pycsp3 import *

nTeams, pv = data
nRounds = 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 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., [23, 34]. 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 automatas) 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 $f : V_p \rightarrow V_t$ 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.

(a) Pattern Graph (b) Target Graph

Figure 3.5: An Instance of the 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 31

```
from pycsp3 import *

n, m, p_edges, t_edges = data

# useful auxiliary structures
p_degrees = [len([edge for edge in p_edges if i in edge]) for i in range(n)]
t_degrees = [len([edge for edge in t_edges if i in edge]) for i in range(m)]
table = {(i, j) for i, j in t_edges} | {(j, i) for i, j in t_edges}
conflicts = [{j for j in range(m) if t_degrees[j] < p_degrees[i]} for i in range(n)]
p_loops =  [i for (i, j) in p_edges if i == j]
t_loops =  [i for (i, j) in t_edges if i == j]
```

```
# 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 p_edges],

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

   # tag(redundant-constraints)
   [x[i] not in t for i, t in enumerate(conflicts)]
)
```

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 (and x `not in` S) where x is a variable of the model and S a set of values. For a negative table constraint, if ever the length of the table is 0, then, no constraint is posted.

3.3 Constraint `regular`

Definition 1 (DFA) *A* deterministic finite automaton *(DFA) is 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,* $\delta : Q \times \Sigma \rightarrow Q$ *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 [11, 28] 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`
$\boldsymbol{x}_0 \boldsymbol{x}_1 \ldots \boldsymbol{x}_{r-1} \in L(A)$

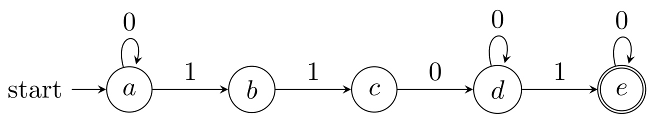
In PyCSP^3, we can directly write `regular` constraints in mathematical forms, by using tuples, automatas and the operator `in`. The scope of a constraint is given by a tuple of variables on the left of the constraining expression and an automaton is given on the right of the constraining expression.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

```
a, b, c, d, e = "a", "b", "c", "d", "e"
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)

satisfy(
   (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`

The constraint `mdd` [13, 14, 15, 27] ensures that the sequence of values assigned to the variables it involves follows a path going from the root of the described MDD (Multi-valued Decision Diagram) to the unique terminal node. Because the graph is directed, acyclic, with only one root node and only one terminal node, we just need to introduce the set of transitions.

Below, $L(M)$ denotes the language recognized by a MDD M.

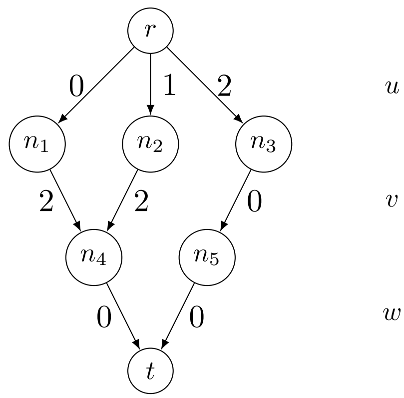
Semantics 5

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

In PyCSP^3, we can directly write `mdd` constraints in mathematical forms, by using tuples, MDDs 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 MDD is given on the right of the constraining expression. MDDs in PyCSP^3 are objects of Class `MDD` that are built by calling the following constructor:

```
def __init__(self, transitions):
```

The named parameter `transitions` is required: this is a list (not a set) of 3-tuples. As said above, the root and terminal nodes (and the full set of states) can be inferred from `transitions`, if the MDD is well constructed.



As an example, the constraint of scope $\langle u, v, w\rangle$ is defined from the simple MDD depicted above (with root node r and terminal node t) as:

```
r, n1, n2, n3, n4, n5, t = "r", "n1", "n2", "n3", "n4", "n5", "t"
tr = [(r,0,n1), (r,1,n2), (r,2,n3), (n1,2,n4), (n2,2,n4), (n3,0,n5), (n4,0,t), (n5,0,t)]

satisfy(
   (u, v, w) in MDD(tr),

   ...
)
```

3.5 Constraint `allDifferent`

The constraint `allDifferent`, see [30, 33, 18], ensures that the variables in a specified list X must all take different values. A variant, called `allDifferentExcept` in the literature [3, 16], enforces variables to take distinct values, except those that are assigned to some specified values (often, the single value 0). This is the role of the set E below.

Semantics 6

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

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

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

The two parameters `term` and `others` are positional, and allow us to pass the terms either in sequence (individually) or under the form of a list. The optional named parameter `excepting` indicates the value (or the set of values) that must be ignored, and the optional named parameter `matrix` indicates if a constraint `allDifferent` must be imposed on both rows and columns of a two-dimensional list (matrix). More accurately, the terms can be given as:

- a list of variables, as in `AllDifferent(x)`
- a sequence of individual variables, as in `AllDifferent(u, v, w)`
- a generator of variables, as in `AllDifferent(x[i] for in range(n) if i%2 > 0)`
- a sequence of individual expressions, as in `AllDifferent(x[1] + 1, x[2] + 2, x[3] + 3)`
- a generator of expressions, as in `AllDifferent(x[i] + i for in range(n))`

Below, we introduce some additional models involving the `allDifferent` constraint.

Send-More-Money. From Wikipedia: Cryptarithmetic is a type of mathematical game consisting of a mathematical equation among unknown numbers, whose digits are represented by letters. The goal is to identify the value of each letter. The classic example, published in the July 1924 issue of Strand Magazine by Henry Dudeney is:

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

A PyCSP^3 model for this specific example is given by:

PyCSP^3 Model 32

```
from pycsp3 import *

# letters[i] is the digit of the ith letter involved in the equation
s, e, n, d, m, o, r, y = letters = VarArray(size=8, dom=range(10))

satisfy(
   # letters are given different values
   AllDifferent(letters),

   # words cannot start with 0
   [s > 0, m > 0],

   # respecting the mathematical equation
      [s, e, n, d] * [1000, 100, 10, 1]
   +  [m, o, r, e] * [1000, 100, 10, 1]
   == [m, o, n, e, y] * [10000, 1000, 100, 10, 1]
)
```

It is important to note that not only variables but also general expressions can be involved in the `allDifferent` constraint, as shown in Section 1.2.1 and the following model.

Costas Arrays. From CSPLib: "A costas array is a pattern of n marks on an $n \times n$ grid, one mark per row and one per column, in which the $n \times (n-1)/2$ (displacement) vectors between the marks are all-different. Such patterns are important as they provide a template for generating radar and sonar signals with ideal ambiguity functions."

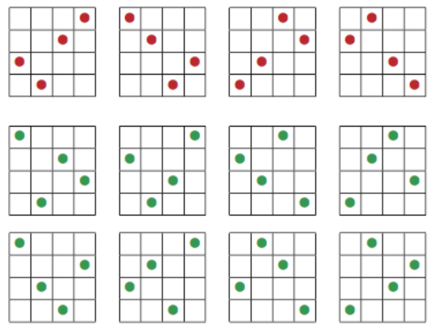


Figure 3.6: The 12 Costas arrays of order 4. (image from commons.wikimedia.org)

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

PyCSP^3 Model 33

```
from pycsp3 import *

n = data

# x[i] is the row where is put the ith mark (on the ith column)
x = VarArray(size=n, dom=range(n))

satisfy(
   # all marks are on different rows (and columns)
   AllDifferent(x),

   # all displacement vectors between the marks must be different
   [AllDifferent(x[i] - x[i + d] for i in range(n - d)) for d in range(1, n - 1)]
)
```

Now, assuming that x is a two-dimensional list of variables, the matrix variant of `allDifferent` is imposed on x by: `AllDifferent(x, matrix=True)`. If $x = [[u_1, u_2, u_3, u_4], [v_1, v_2, v_3, v_4], [w_1, w_2, w_3, w_4]]$, then the posted constraint is equivalent to having posted:

- `AllDifferent`(u_1, u_2, u_3, u_4)
- `AllDifferent`(v_1, v_2, v_3, v_4)
- `AllDifferent`(w_1, w_2, w_3, w_4)
- `AllDifferent`(u_1, v_1, w_1)
- `AllDifferent`(u_2, v_2, w_2)
- `AllDifferent`(u_3, v_3, w_3)
- `AllDifferent`(u_4, v_4, w_4)

The matrix variant of `allDifferent` was introduced in Section 1.3.1. Here is another illustration.

Futoshiki. From Wikipedia: “Futoshiki is a logic puzzle game from Japan, which was developed by Tamaki Seto in 2001. The puzzle is played on a square grid, and the objective is to place the numbers such that each row and column contains only one of each digit. Some digits may be given at the start, and inequality constraints are initially specified between some of the squares, such that one must be higher or lower than its neighbor.”

(a) Puzzle (b) Solution

Figure 3.7: Solving a Futoshiki Puzzle. (images from commons.wikimedia.org)

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

```
{
  "size": 3,
  "nbHints": [{"row":0, "col":0, "number":2}],
  "opHints": [
    {"row":0, "col":1, "lessThan":true, "horizontal":true},
    {"row":2, "col":0, "lessThan":true, "horizontal":true}
  ]
}
```

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

PyCSP^3 Model 34

```
from pycsp3 import *

n, nbHints, opHints = data  # n is the order of the grid

# x[i][j] is the number put at row i and column j
x = VarArray(size=[n, n], dom=range(1, n + 1))

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

   # respecting number hints
   [x[i][j] == k for (i, j, k) in nbHints],

   # respecting operator hints
   [y < z if lt else y > z
     for (y, z, lt) in [(x[i][j], x[i][j + 1] if hr else x[i + 1][j], lt)
       for (i, j, lt, hr) in opHints]]
)
```

Because objects from the JSON file are automatically converted to named tuples, note how we can use tuple unpacking when iterating overs lists of such objects.

Here is now an illustration concerning the "except" variant of `allDifferent`.

Progressive Party. This problem will be introduced in Section 3.17. Here is a snippet of the PyCSP^3 model:

```
...

# 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))

satisfy(
   ...

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

   ...
}
```

Because the crew can stay several periods on his boat, while visiting different boats on other periods, we need `allDifferent` with the named parameter `excepting`.

3.6 Constraint `allDifferentList`

The constraint `allDifferentList` admits as parameters two (or more) lists of integer variables, and ensures that the tuple of values taken by variables of the first list is different from the tuple of values taken by variables of the second list. If more than two lists are given, all tuples must be different. A variant enforces tuples to take distinct values, except those that are assigned to some specified tuples (often, the single tuple containing only 0).

Semantics 7

`allDifferentList`$(\mathcal{X}, E)$, with $\mathcal{X} = \langle X_1, X_2, \ldots \rangle$, E the set of discarded tuples, iff
$\forall (i,j) : 1 \leq i < j \leq |\mathcal{X}|, \boldsymbol{X}_i \neq \boldsymbol{X}_j \vee \boldsymbol{X}_i \in E \vee \boldsymbol{X}_j \in E$
`allDifferentList`$(\mathcal{X})$ iff `allDifferentList`$(\mathcal{X}, \emptyset)$

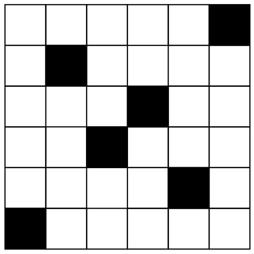
Prerequisite : $|\mathcal{X}| \geq 2 \wedge \forall i : 1 \leq i < |\mathcal{X}|, |X_i| = |X_{i+1}| \geq 2 \wedge \forall \tau \in E, |\tau| = |X_1|$

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

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

The two parameters `term` and `others` are positional, and allow us to pass the terms either in sequence (individually) or under the form of a matrix. The optional named parameter `excepting` indicates the tuple (or the set of tuples) that must be ignored.

Crossword. "Given a grid with imposed black cells (spots) and a dictionary, the problem is to fulfill the grid with the words contained in the dictionary." An illustration is given by Figure 3.8.



(a) Crossword Grid

A	L	O	H	A	■
X	■	R	I	C	H
I	C	E	■	H	A
O	R	■	W	E	I
M	A	M	A	■	F
■	G	E	N	O	A

(b) Solution

Figure 3.8: Making a Crossword Puzzle.

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

```
{
  "spots": [
    [0,0,0,0,0,1],
    [0,1,0,0,0,0],
    [0,0,0,1,0,0],
    [0,0,1,0,0,0],
    [0,0,0,0,1,0],
    [1,0,0,0,0,0]],
  "dictFileName": "ogd"
}
```

The grid is specified by the field `spots` of the root object in the JSON file; when present, the value 1 means the presence of a spot (black cell). The name of the dictionary to be used is also given (as it is not reasonable to include it in the JSON file).

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

PyCSP^3 Model 35

```
from pycsp3 import *

spots, dict_name = data
words = dict()  # we load/build the dictionary of words
for line in open(dict_name):
  code = alphabet_positions(line.strip().lower())
  words.setdefault(len(code), []).append(code)

def find_holes(tab, transposed):
  def build_hole(row, col, length, horizontal):
    return Hole(row, slice(col, col + length), length) if horizontal
           else Hole(slice(col, col + length), row, length)

  Hole = namedtuple("Hole", "i j r")  # i and j are indexes (one being a slice)
  p, q = len(tab), len(tab[0])
  t = []
  for i in range(p):
    start = -1
    for j in range(q):
      if tab[i][j] == 1:
        if start != -1 and j - start >= 2:
          t.append(build_hole(i, start, j - start, not transposed))
        start = -1
      elif start == -1:
        start = j
      elif j == q - 1 and q - start >= 2:
        t.append(build_hole(i, start, q - start, not transposed))
  return t
```

```
holes = find_holes(spots, False) + find_holes(transpose(spots), True)
arities = sorted(set(arity for (_, _, arity) in holes))
n, m, nHoles = len(spots), len(spots[0]), len(holes)

#  x[i][j] is the letter, number from 0 to 25, at row i and column j (when no spot)
x = VarArray(size=[n, m], dom=lambda i, j: range(26) if spots[i][j] == 0 else None)

satisfy(
   # fill the grid with words
   [x[i, j] in words[r] for (i, j, r) in holes],

   # tag(distinct-words)
   [AllDifferentList(x[i, j] for (i, j, r) in holes if r == arity)
     for arity in arities]
)
```

3.7 Constraint `allEqual`

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

Semantics 8

`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 [35] 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: $(x_0 + 1 = x_{n-1}) \vee (x_0 = x_{n-1} = d-1)$

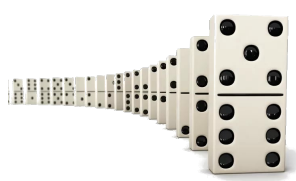


Figure 3.9: Filtering as a Domino (cascade) effect. (image from pngimg.com)

Those who are interested in the way domains of variables can be filtered (i.e., reduced) in this problem will observe a kind of Domino (cascade) effect [35, 24]. A PyCSP^3 model of this problem is given by the following file 'Domino.py':

PyCSP^3 Model 36

```
from pycsp3 import *

n, d = data

# 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]) & (x[0] == 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.

3.8 Constraints `increasing` and `decreasing`

The constraint `ordered` ensures that the variables of a specified list of variables X are ordered in sequence, according to a specified relational operator $\odot \in \{<, \leq, \geq, >\}$. An optional list of integers or variables L indicates the minimum distance between any two successive variables of X.

Semantics 9

`ordered`$(X, L, \odot)$, `with` $X = \langle x_0, x_1, \ldots\rangle$, $L = \langle l_0, l_1, \ldots\rangle$ `and` $\odot \in \{<, \leq, \geq, >\}$, `iff`
$\forall i : 0 \leq i < |X| - 1, \boldsymbol{x}_i + l_i \odot \boldsymbol{x}_{i+1}$
`ordered`$(X, \odot)$, `with` $X = \langle x_0, x_1, \ldots\rangle$ `and` $\odot \in \{<, \leq, \geq, >\}$, `iff`
$\forall i : 0 \leq i < |X| - 1, \boldsymbol{x}_i \odot \boldsymbol{x}_{i+1}$

Prerequisite : $|X| = |L| + 1$

In PyCSP^3, to post a constraint `ordered`, we must call either the function `Increasing`() or the function `Decreasing`(), whose signatures are:

```
def Increasing(term, *others, strict=False, lengths=None):
def Decreasing(term, *others, strict=False, lengths=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 `strict` indicates if the relation must be strict or not, and the optional named parameter `lengths` is for specifying minimum distances. In other words, assuming that $x = [u, v, w]$ is a simple list of variables, ordering variables of x can be imposed by:

- `Increasing(x, strict=True)`
  ensuring $u < v < w$
- `Increasing(x)`
  ensuring $u \leq v \leq w$
- `Decreasing(x)`
  ensuring $u \geq v \geq w$

- Decreasing(x, strict=True)
  ensuring $u > v > w$

The constraints **increasing** and **decreasing** are mainly an ease of use, as it is possible to post equivalent **intension** constraints. For example, `Increasing(x,strict=True)` can be equivalently written as:

```
[x[i] < x[i+1] for i in range(len(x)-1)]
```

Steiner Triple Systems. From CSPLib: "The ternary Steiner problem of order n consists of finding a set of $n \times (n1)/6$ triples of distinct integer elements in $\{1, 2, \ldots, n\}$ such that any two triples have at most one common element. It is a hypergraph problem coming from combinatorial mathematics where n modulo 6 has to be equal to 1 or 3. One possible solution for $n = 7$ is $\{\{1, 2, 3\}, \{1, 4, 5\}, \{1, 6, 7\}, \{2, 4, 6\}, \{2, 5, 7\}, \{3, 4, 7\}, \{3, 5, 6\}\}$. This is a particular case of the more general Steiner system."

A PyCSP^3 model of this problem is given by the following file 'Steiner3.py':

PyCSP^3 Model 37

```
from pycsp3 import *

n = data
nTriples = (n * (n - 1)) // 6

table = {(i1, i2, i3, j1, j2, j3)
          for (i1, i2, i3, j1, j2, j3) in product(range(1, n + 1), repeat=6)
            if different_values(i1, i2, i3) and different_values(j1, j2, j3)
               and len({i for i in {i1, i2, i3} if i in {j1, j2, j3}}) <= 1}

# x[i] is the ith triplet of value
x = VarArray(size=[nTriples, 3], dom=range(1, n + 1))

satisfy(
    # each triple must be formed of strictly increasing integers
    [Increasing(triple, strict=True) for triple in x],

    # each pair of triples must share at most one value
    [(triple1 + triple2) in table for (triple1, triple2) in combinations(x, 2)]
)
```

3.9 Constraints `lexIncreasing` and `lexDecreasing`

The constraint `ordered` can be naturally lifted to lists, by considering the lexicographic order. Because this constraint is very popular, it is called `lex`, instead of `ordered` over lists of integer variables. The constraint `lex`, see [10, 17], ensures that the tuple formed by the values assigned to the variables of a first specified list X_1 is related to the tuple formed by the values assigned to the variables of a second specified list X_2 with respect to a specified lexicographic order operator $\odot \in \{<_{lex}, \leq_{lex}, \geq_{lex}, >_{lex}\}$. If more than two lists of variables are specified, the entire sequence of tuples must be ordered; this captures then `lexChain` [9].

Semantics 10

`lex`$(\mathcal{X}, \odot)$, `with` $\mathcal{X} = \langle X_0, X_1, \ldots \rangle$ `and` $\odot \in \{<_{lex}, \leq_{lex}, \geq_{lex}, >_{lex}\}$, `iff`
$\forall i : 0 \leq i < |\mathcal{X}| - 1, \boldsymbol{X}_i \odot \boldsymbol{X}_{i+1}$

Prerequisite : $|\mathcal{X}| \geq 2 \wedge \forall i : 0 \leq i < |\mathcal{X}| - 1, |X_i| = |X_{i+1}| \geq 2$

In PyCSP^3, to post a constraint `lex`, we must call either the function `LexIncreasing`() or the function `lexDecreasing`(), whose signatures are:

```
def LexIncreasing(term, *others, strict=False, matrix=False):
def LexDecreasing(term, *others, strict=False, matrix=False):
```

The two parameters `term` and `others` are positional, and allow us to pass the lists either in sequence (individually) or under the form of a two-dimensional list. The optional named parameter `strict` indicates if the relation must be strict or not, and the optional named parameter `matrix` indicates if a lexicographic order must be imposed on both rows and columns of a two-dimensional list (matrix). In other words, assuming that x, y and z are simple lists of variables, ordering lexicographically x, y and z can be imposed by:

- `LexIncreasing(x1, x2, x3, strict=True)`
  ensuring $x <_{lex} y <_{lex} z$
- `LexIncreasing(x1, x2, x3)`
  ensuring $x \leq_{lex} y \leq_{lex} z$
- `LexDecreasing(x1, x2, x3)`
  ensuring $x \geq_{lex} y \geq_{lex} z$
- `LexDecreasing(x1, x2, x3, strict=True)`
  ensuring $x >_{lex} y >_{lex} z$

Now, assuming that x is a two-dimensional list of variables, the matrix variant of `lex` with $\leq_{lex}$ (for example) as operator is imposed on x by: `LexIncreasing(x, matrix=True)`. If $x = [[p, q, r], [u, v, w]]$, then the posted constraint is equivalent to having posted:

- $(p, q, r) \leq_{lex} (u, v, w)$
- $(p, u) \leq_{lex} (q, v) \leq_{lex} (r, w)$

Figure 3.10: A golfer who apparently needs socialization. (image from www.publicdomainpictures.net)

Social Golfers. "The coordinator of a local golf club has come to you with the following problem. In their club, there are 32 social golfers, each of whom play golf once a week, and always in groups of 4. They would like you to come up with a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion. The problem can easily be generalized to that of scheduling G groups of K golfers over at most W weeks, such that no golfer plays in the same group as any other golfer twice (i.e. maximum socialisation is achieved). For the original problem, the values of G and K are respectively 8 and 4." See CSPLib.

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

PyCSP^3 Model 38

```
from pycsp3 import *

nGroups, size, nWeeks = data
nPlayers = nGroups * size

# g[w][p] is the group admitting on week w the player p
g = VarArray(size=[nWeeks, nPlayers], dom=range(nGroups))

satisfy(
   # ensuring that two players don't meet more than one time
   [(g[w1][p1] != g[w1][p2]) | (g[w2][p1] != g[w2][p2])
     for w1, w2 in combinations(nWeeks, 2) for p1, p2 in combinations(nPlayers, 2)],

   # respecting the size of the groups
   [Cardinality(g[w], occurrences={i: size for i in range(nGroups)})
     for w in range(nWeeks)],

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

We have the guarantee of keeping at least one solution if the instance is satisfiable, when the matrix `lex` constraint is posted.

3.10 Constraint `sum`

The constraint `sum` is one of the most important constraint. This constraint may involve (integer or variable) coefficients, and is subject to a numerical condition $(\odot, k)$. For example, a form of `sum`, sometimes called `subset-sum` or `knapsack` [32, 29] involves the operator `in`, and ensures that the computed sum belongs to a specified interval. Below, we introduce the semantics while considering a main list X of variables and a list C of coefficients:

Semantics 11

`sum`$(X, C, (\odot, k))$, with $X = \langle x_0, x_1, \ldots \rangle$, and $C = \langle c_0, c_1, \ldots \rangle$, iff
$(\sum_{i=0}^{|X|-1} \boldsymbol{c}_i \times \boldsymbol{x}_i) \odot \boldsymbol{k}$

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

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

```
def Sum(term, *others):
```

The two parameters `term` and `others` are positional, and allow us to pass the terms either in sequence (individually) or under the form of a list. More accurately, the terms can be given as:

- a list of variables, as in `Sum(x)`

- ◦ a sequence of individual variables, as in `Sum(u, v, w)`
- ◦ a generator of variables, as in `Sum(x[i] for in range(n) if i%2 > 0)`
- ◦ a generator of variables, with coefficients, as in `Sum(x[i] * costs[i] for in range(n))`
- ◦ a generator of expressions, as in `Sum(x[i] > 0 for in range(n))`
- ◦ a generator of expressions, with coefficients, as in `Sum((x[i] + y[i]) * costs[i] for in range(n))`

Note that arguments are flattened, meaning that variables (and expressions) are collected from arguments to form a simple list even if multi-dimensional structures (lists) are involved, and while discarding any occurrence of the value `None`. For example, flattening `[ [u, v], [None, w]]` gives `[u, v, w]`.

The object obtained when calling `Sum()` must be restricted by a condition (typically, defined by a relational operator and a limit).

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

```
satisfy(
   ...

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

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

The first `sum` constraint involves a simple list x of variables whereas the second one involves terms that are products of variables and coefficients.

Importantly, it is possible to combine several objects `Sum` with operators + and − (and to compare them, which is equivalent to a subtraction). This is illustrated below, with a general model for crypto-arithmetic puzzles (in Section 3.5, we introduced a specific model dedicated to 'send+more=money').

Crypto Puzzle. In crypto-arithmetic problems, digits (values between 0 and 9) are represented by letters. Different letters stand for different digits, and different occurrences of the same letter denote the same digit. The problem is then represented as an arithmetic operation between words. The task is to find out which letter stands for which digit, so that the result of the given arithmetic operation is true.

For example,

```
      N O            C R O S S           D O N A L D
+     N O        +     R O A D S     +   G E R A L D
=   Y E S        =   D A N G E R     =   R O B E R T
```

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

PyCSP^3 Model 39

```
word1, word2, word3 = words = [w.lower() for w in data]
letters = set(alphabet_positions(word1 + word2 + word3))
n = len(word1) ; assert len(word2) == n and len(word3) in {n, n + 1}

# x[i] is the value assigned to the ith letter (if present) of the alphabet
x = VarArray(size=26, dom=lambda i: range(10) if i in letters else None)

# auxiliary lists of variables associated with the three words
x1, x2, x3 = [[x[i] for i in reversed(alphabet_positions(word))] for word in words]

satisfy(
    # all letters must be assigned different values
    AllDifferent(x),

    # the most significant letter of each word cannot be equal to 0
    [x1[-1] != 0, x2[-1] != 0, x3[-1] != 0]

    # ensuring the crypto-arithmetic sum
    Sum((x1[i] + x2[i]) * 10 ** i for i in range(n))
      == Sum(x3[i] * 10 ** i for i in range(len(x3)))
)
```

The PyCSP^3 function `alphabet_positions()` returns a tuple composed with the position in the alphabet of all letters of a specified string. For example, `alphabet_positions("about")` returns `(0, 1, 14, 20, 19)`. Note how two objects `Sum` are involved. Of course the crypto-arithmetic sum could also have been written as:

```
Sum((x1[i] + x2[i]) * 10 ** i for i in range(n))
  - Sum(x3[i] * 10 ** i for i in range(len(x3))) == 0
```

To well understand the way the constraint `sum` is constructed, note that executing:

```
python3 CryptoPuzzle.py -data=[SEND,MORE,MONEY]
```

yields the following XCSP^3 file:

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i] is the value assigned to the ith letter (if present) of
        the alphabet" size="[26]"> 0..9 </array>
  </variables>
  <constraints>
    <allDifferent note="all letters must be assigned different values">
      x[3..4] x[12..14] x[17..18] x[24]
    </allDifferent>
    <group note="the most significant letter of each word cannot be equal to 0">
      <intension> ne(%0,0) </intension>
      <args> x[18] </args>
      <args> x[12] </args>
      <args> x[12] </args>
    </group>
    <sum note="ensuring the crypto-arithmetic sum">
      <list> add(x[3],x[4]) add(x[13],x[17]) add(x[4],x[14]) add(x[18],x[12])
             x[24] x[4] x[13..14] x[12] </list>
      <coeffs> 1 10 100 1000 -1 -10 -100 -1000 -10000 </coeffs>
      <condition> (eq,0) </condition>
    </sum>
  </constraints>
</instance>
```

Finally, it is possible to use dot product to build a weighted sum. It means that it suffices to use the operator $*$ between two lists involving variables, integers or expressions to obtain an object `Sum` as e.g., in `[u, v, w] * [2, 4, 3]` which represents $u*2+v*4+w*3$. An illustration is given below.

Template Design. From CSPLib: "This problem arises from a colour printing firm which produces a variety of products from thin board, including cartons for human and animal food and magazine inserts. Food products, for example, are often marketed as a basic brand with several variations (typically flavours). Packaging for such variations usually has the same overall design, in particular the same size and shape, but differs in a small proportion of the text displayed and/or in colour. For instance, two variations of a cat food carton may differ only in that on one is printed Chicken Flavour on a blue background whereas the other has Rabbit Flavour printed on a green background. A typical order is for a variety of quantities of several design variations. Because each variation is identical in dimension, we know in advance exactly how many items can be printed on each mother sheet of board, whose dimensions are largely determined by the dimensions of the printing machinery. Each mother sheet is printed from a template, consisting of a thin aluminium sheet on which the design for several of the variations is etched. Each design of carton is made from an identically sized and shaped piece of board. Several cartons can be printed on each mother sheet (in slots), and several different designs can be printed at once, on the same mother sheet. The problem is to decide, firstly, how many distinct templates to produce, and secondly, which variations, and how many copies of each, to include on each template, in order to minimize the amount of waste produced. " More details, and an example, are given on CSPLib.

Figure 3.11: Cat Food Cartons. (image from www.vecteezy.com)

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

```
{
  "nSlots": 9,
  "demands": [250, 255, 260, 500, 500, 800, 1100]
}
```

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

PyCSP^3 Model 40

```
from pycsp3 import *
from math import ceil, floor

nSlots, demands = data
nTemplates = nVariations = len(demands)

def variation_interval(v):
    return range(ceil(demands[v] * 0.95), floor(demands[v] * 1.1) + 1)

# d[i][j] is the number of occurrences of the jth variation on the ith template
d = VarArray(size=[nTemplates, nVariations], dom=range(nSlots + 1))

# p[i] is the number of printings of the ith template
p = VarArray(size=nTemplates, dom=range(max(demands) + 1))
```

```
satisfy(
   # all slots of all templates are used
   [Sum(d[i]) == nSlots for i in range(nTemplates)],

   # respecting printing bounds for each variation
   [p * d[:, j] in variation_interval(j) for j in range(nVariations)]
)

minimize(
   # minimizing the number of used templates
   Sum(p[i] > 0 for i in range(nTemplates))
)
```

The two arguments of `satisfy()` correspond to two lists of `sum` constraints; the second list involves dot products, each one built from the array (list) of variables p and the jth column of the two-dimensional array (list) d, and imposed to belong to a certain interval.

3.11 Constraint `count`

The constraint `count`[1], imposes that the number of variables from a specified list of variables X that take their values from a specified set V respects a numerical condition $(\odot, k)$. This constraint captures known constraints (usually) called `atLeast`, `atMost`, `exactly` and `among`. To simplify, we assume for the semantics that V is a set of integer values.

Semantics 12

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

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

```
def Count(term, *others, value=None, values=None):
```

The two parameters `term` and `others` are positional, and allow us to pass the main list of variables X either in sequence (individually) or under the form of a list. The two named parameters allow us to specify either a single value (unique target for counting) or a set of values. Exactly one of these two parameters must be different from `None`. Assuming that x is a list of variables, here are a few examples:

- `Count(x, values={1, 5, 8}) == k`
  stands for 'k variables from x must take their values *among* those in $\{1, 5, 8\}$'
- `Count(x, value=0) > 1`
  stands for '*at least* 2 variables from x must be assigned to the value 0'
- `Count(x, value=1) <= k`
  stands for '*at most* k variables from x must be assigned to the value 1'
- `Count(x, value=z) == k`
  stands for '*exactly* k variables from x must be assigned to the value z'

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

[1] initially introduced in CHIP [2] and Sicstus [12]

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

    ...
)
```

Each `count` constraint imposes that the number of variables in w that take the value j is at most equal to the capacity of the jth warehouse.

Pizza Voucher Problem. From the Intelligent Systems CMPT 417 course at Simon Fraser University. "The problem arises in the University College Cork student dorms. There is a large order of pizzas for a party, and many of the students have vouchers for acquiring discounts in purchasing pizzas. A voucher is a pair of numbers e.g. $(2, 4)$, which means if you pay for 2 pizzas then you can obtain for free up to 4 pizzas as long as they each cost no more than the cheapest of the 2 pizzas you paid for. Similarly a voucher $(3, 2)$ means that if you pay for 3 pizzas you can get up to 2 pizzas for free as long as they each cost no more than the cheapest of the 3 pizzas you paid for. The aim is to obtain all the ordered pizzas for the least possible cost. Note that not all vouchers need to be used."

Figure 3.12: A Nice Pizza Slice. (image from freesvg.org)

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

```
{
  "pizzaPrices": [50, 60, 90, 70, 80, 100, 20, 30, 40, 10],
  "vouchers":[
    {"payPart":1,"freePart":2},
    {"payPart":2,"freePart":3},
    ...
  ]
}
```

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

PyCSP^3 Model 41

```
from pycsp3 import *

prices, vouchers = data
nPizzas, nVouchers = len(prices), len(vouchers)

# v[i] is the voucher used for the ith pizza. 0 means that no voucher is used.
# A negative (resp., positive) value i means that the ith pizza contributes
# to the the pay (resp., free) part of voucher |i|.
v = VarArray(size=nPizzas, dom=range(-nVouchers, nVouchers + 1))

# p[i] is the number of paid pizzas wrt the ith voucher
p = VarArray(size=nVouchers, dom=lambda i: {0, vouchers[i].payPart})

# f[i] is the number of free pizzas wrt the ith voucher
f = VarArray(size=nVouchers, dom=lambda i: range(vouchers[i].freePart + 1))
```

```
satisfy(
   # counting paid pizzas
   [Count(v, value=-i - 1) == p[i] for i in range(nVouchers)],

   # counting free pizzas
   [Count(v, value=i + 1) == f[i] for i in range(nVouchers)],

   # a voucher, if used, must contribute to have at least one free pizza.
   [iff(f[i] == 0, p[i] != vouchers[i].payPart) for i in range(nVouchers)],

   # a free pizza must be cheaper than any pizza paid wrt the used voucher
   [imply(v[i] < 0, v[i] != -v[j]) for i in range(nPizzas)
     for j in range(nPizzas) if i != j and prices[i] < prices[j]]
)

minimize(
   # minimizing summed up costs of pizzas
   Sum((v[i] <= 0) * prices[i] for i in range(nPizzas))
)
```

3.12 Constraint `nValues`

The constraint `nValues` [4], 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` [4] discards some specified values of a set E (often, the single value 0).

Semantics 13

`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 PyCSP^3, 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`.

RLFAP. This problem was introduced in Section 2.3. The function `NValues`() was used to specify the objective of one variant of the problem.

3.13 Constraint `cardinality`

The constraint `cardinality`, also called `globalCardinality` or `gcc` in the literature, see [31, 22], 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 14

`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 optional 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 dictionary.

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 42

```
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 i in alphabet_positions("".join(words)) else None)

satisfy(
   # the four letters of each word appears on different cubes
   [AllDifferent(x[i] for i in 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 `alphabet_positions()` returns a tuple composed with the position in the alphabet of all letters of a specified string. For example, `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.

"

Figure 3.13: Sports Scheduling. (image from commons.wikimedia.org)

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

PyCSP^3 Model 43

```
from pycsp3 import *

nTeams = data
nWeeks, nPeriods, nMatches = nTeams - 1, nTeams // 2, (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, t2 in combinations(range(nTeams), 2)}
```

```
# 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.

3.14 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 15

`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 44

```
from pycsp3 import *

orders = data
n, m = len(orders), len(orders[0])  # n orders (customers), 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})

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)],
)

minimize(
   # minimizing the number of stacks that are simultaneously open
   Maximum(Sum(o[:, t]) for t in range(m))
)
```

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.

3.15 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 16

`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.

3.16 Constraint `element`

The constraint `element` [21] 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 17

`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 45

```
from pycsp3 import *

alice, bob, sascha = persons = 0, 1, 2

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

# liking[i][j] is 1 iff the ith guy likes the jth guy
liking = 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
   liking[culprit][alice] == 1,

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

   # nobody is taller than himself
   [taller[p][p] == 0 for p in persons],

   # the ith guy is taller than the jth guy iff the reverse is not true
   [taller[p1][p2] != taller[p2][p1] for p1 in persons for p2 in persons if p1 != p2],

   # Bob likes no one that Alice likes
   [imply(liking[alice][p], ~liking[bob][p]) for p in persons],

   # Alice likes everybody except Bob
   [liking[alice][p] == 1 for p in persons if p != bob],

   # Sascha likes everyone that Alice likes
   [imply(liking[alice][p], liking[sascha][p]) for p in persons],

   # nobody likes everyone
   [Count(liking[p], value=0) >= 1 for p in persons]
)
```

The variant 2 of `element` is illustrated by:

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

as it basically encodes “the variable at index `culprit` in the column 0 (alice) of the 2-dimensional array of variables `liking` 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]".

Interestingly, it is also possible to use a variant of `element` on matrices, i.e., by using two indexes given by integer variables. The semantics is `M[i][j] = v`, or equivalently:

Semantics 18

`element`$(\mathcal{M}, \langle i, j\rangle, v)$, `with` $\mathcal{M} = [\langle x_{1,1}, x_{1,2}, \ldots, x_{1,m}\rangle, \langle x_{2,1}, x_{2,2}, \ldots, x_{2,m}\rangle, \ldots]$, `iff` $\boldsymbol{x_{i,j} = v}$

It is important to note that i and j must be two integer variables (and not constants). In Python, to post an `element` constraint on matrices, 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` on matrices:

- variant 1: M is a matrix of variables, i and j are integer variables and v is an integer variable
- variant 2: M is a matrix of variables, i and j are integer variables and v is an integer (constant)
- variant 3: M is a matrix of integers, i and j are integer variables and v is an integer variable

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

Quasigroup Existence. From CSPLib: "A quasigroup of order n is a $n \times n$ multiplication table in which each element occurs once in every row and column (i.e., is a Latin square), while satisfying some specific properties. Hence, the result $a * b$ of applying the multiplication operator $*$ on a (left operand) and b (right operand) is given by the value in the table at row a and column b. Classical variants of quasigroup existence correspond to taking into account the following properties:

- QG3: quasigroups for which $(a * b) * (b * a) = a$
- QG4: quasigroups for which $(b * a) * (a * b) = a$
- QG5: quasigroups for which $((b * a) * b) * b = a$
- QG6: quasigroups for which $(a * b) * b = a * (a * b)$
- QG7: quasigroups for which $(b * a) * b = a * (b * a)$

For each of these problems, we may additionally demand that the quasigroup is idempotent. That is, $a * a = a$ for every element a."

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

PyCSP^3 Model 46

```
from pycsp3 import *

n = data

# x[i][j] is the value at row i and column j of the quasi-group
x = VarArray(size=[n, n], dom=range(n))

satisfy(
   # ensuring a Latin square
   AllDifferent(x, matrix=True),
```

```
    # ensuring idempotence  tag(idempotence)
    [x[i][i] == i for i in range(n)]
)

if variant("v3"):
    satisfy(
        x[x[i][j], x[j][i]] == i for i in range(n) for j in range(n)
    )
elif variant("v4"):
    satisfy(
        x[x[j, i], x[i, j]] == i for i in range(n) for j in range(n)
    )
elif variant("v5"):
    satisfy(
        x[x[x[j, i], j], j] == i for i in range(n) for j in range(n)
    )
elif variant("v6"):
    satisfy(
        x[x[i, j], j] == x[i, x[i, j]] for i in range(n) for j in range(n)
   )
elif variant("v7"):
    satisfy(
        x[x[j, i], j] == x[i, x[j, i]] for i in range(n) for j in range(n)
    )
```

The variant 2 of `element` on matrices is illustrated by:

```
x[x[i][j], x[j][i]] == i
```

as it basically encodes "the variable in the matrix x at row index `x[i][j]` (a variable) and column index `x[j][i]` (a variable) must be equal to the integer i". Note how we can write complex operations involving several (partial forms of) `element` constraints; when compiling, auxiliary variables may possibly be introduced (the interested reader can look at the generated XCSP^3 files).

Traveling Salesman Problem (TSP). From Wikipedia: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"

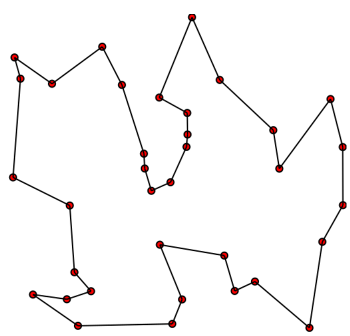


Figure 3.14: A Solution for a TSP instance. (image from commons.wikimedia.org)

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


```
{
  "distances": [
    [0, 5, 6, 6, 6],
    [5, 0, 9, 8, 4],
    [6, 9, 0, 1, 7],
```

```
        [6, 8, 1, 0, 6],
        [6, 4, 7, 6, 0]
      ]
  }
```

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

PyCSP^3 Model 47

```
from pycsp3 import *

distances = data
nCities = len(distances)

# c[i] is the ith city of the tour
c = VarArray(size=nCities, dom=range(nCities))

# d[i] is the distance between the cities i and i+1 chosen in the tour
d = VarArray(size=nCities, dom={v for row in distances for v in row})

satisfy(
   # Visiting each city only once
   AllDifferent(c)
)

if not variant():
   satisfy(
      # computing the distance between any two successive cities in the tour
      distances[c[i]][c[(i + 1) % nCities]] == d[i] for i in range(nCities)
   )

elif variant("table"):
   table = {(i, j, distances[i][j]) for i in range(nCities) for j in range(nCities)}

   satisfy(
      # computing the distance between any two successive cities in the tour
      (c[i], c[(i + 1) % nCities], d[i]) in table for i in range(nCities)
   )

minimize(
   # minimizing the travelled distance
   Sum(d)
)
```

The variant 3 of `element` on matrices is illustrated by:

```
distances[c[i]][c[(i + 1) % nCities]] == d[i]
```

as it basically encodes "the integer in the matrix `distances` at row index `c[i]` (a variable) and column index `c[(i + 1) % nCities]` (a variable) must be equal to the variable `d[i]`". The variant "table" shows which ternary table constraints are equivalent to the `element` constraints on matrices (of integers).

3.17 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 19

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 20

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 21

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 22

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. 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."

Figure 3.15: Progressive Party at a Yacht Club. (image from pngimg.com)

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':

PyCSP^3 Model 48

```
from pycsp3 import *

nPeriods, boats = data
nBoats = len(boats)
capacities, crews = zip(*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 <= capacities[b] 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, b2 in combinations(range(nBoats), 2)]
}

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 <= capacities[b]
  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.

3.18 Constraint `noOverlap`

We start with the one dimensional form of `noOverlap` [22] that corresponds to `disjunctive` [8] 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 23

`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. From 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."

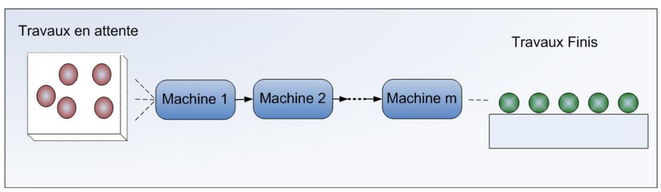


Figure 3.16: Flow Shop Scheduling. (image from commons.wikimedia.org)

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 49

```
from pycsp3 import *

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

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

satisfy(
   # operations must be ordered on each job
   [Increasing(s[i], lengths=durations[i]) 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(s[i][- 1] + durations[i][- 1] for i in range(n))
)
```

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 24

`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 50

```
from pycsp3 import *

width, height = data.container
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
)
```

3.19 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 25

`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 optimization problem where the makespan (i.e., the completion time of the last activity) is minimized." 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 51

```
from pycsp3 import *

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

def cumulative_for(k):
   origins, lengths, heights = zip(*[(s[i], duration, quantities[k])
     for i, (duration, _, quantities) in enumerate(jobs) if quantities[k] > 0])
   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] + duration <= s[j] for i, (duration, successors, _) in enumerate(jobs)
     for j in 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.

3.20 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 26

`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
    },
    ...
  ]
}
```

Figure 3.17: Finding the Best Path for Mario. (image from pngimg.com)

A PyCSP^3 model[2] of this problem is given by the following file 'Mario.py':

PyCSP^3 Model 52

```
from pycsp3 import *

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

satisfy(
   # fuel consumption at each step
   [fuels[i][s[i]] == f[i] for i in range(nHouses)],

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

   # 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((s[i] != i) * houses[i].gold for i in range(nHouses)
     if i not in {marioHouse, luigiHouse})
)
```

[2]This model is inspired from the one proposed by Ollagnier and Fages for the 2013 Minizinc Competition.

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 would 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)],
```

3.21 Meta-Constraint `slide`

A general mechanism, or meta-constraint, that is useful to post constraints on sequences of variables is `slide` [5]. The scheme `slide` ensures that a given constraint is enforced all along a sequence of variables. To represent such sliding constraints in XCSP^3, we simply build an element `<slide>` containing a constraint template (for example, one for `<extension>` or `<intension>`) to indicate the abstract (parameterized) form of the constraint to be slided, preceded by an element `<list>` that indicates the sequence of variables on which the constraint must slide.

For the semantics, we consider that $\texttt{ctr}(\%0,\ldots,\%q-1)$ denotes the template of the constraint `ctr` of arity q, and that $\texttt{slide}^{circ}$ means the circular form of `slide`

Semantics 27

$\texttt{slide}(X, \texttt{ctr}(\%0,\ldots,\%q-1))$, with $X = \langle x_0, x_1, \ldots\rangle$, iff
$\quad \forall i : 0 \leq i \leq |X| - q, \texttt{ctr}(x_i, x_{i+1}, \ldots, x_{i+q-1})$
$\texttt{slide}(X, os, \texttt{ctr}(\%0,\ldots,\%q-1))$, with an offset os, iff
$\quad \forall i : 0 \leq i \leq (|X| - q)/os, \texttt{ctr}(x_{i\times os}, x_{i\times os+1}, \ldots, x_{i\times os+q-1})$
$\texttt{slide}^{circ}(X, \texttt{ctr}(\%0,\ldots,\%q-1))$ iff
$\quad \forall i : 0 \leq i \leq |X| - q + 1, \texttt{ctr}(x_i, x_{i+1} \ldots, x_{(i+q-1)\%|X|})$

In PyCSP^3, to post a (meta-)constraint `slide`, we must call the function `Slide`() whose signature is:

```
def Slide(*args):
```

The specified arguments must correspond to a list (or a set, or even a generator) of sliding constraints. The PyCSP^3 compiler will then attempt to build the XCSP^3 sliding form.

It is important to note that `slide` is interesting only if reasoning with the meta-constraint is stronger than reasoning with each constraint individually. It is also interesting for generating compacter XCSP^3 files (however, you can simply use the option `-recognizeSlides`). An illustration is given in Section 1.3.3.

Chapter 4

Frequently Asked Questions

This chapter will contain frequently asked questions. It needs to be extended.

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)
```

Q. Is it possible to use the PyCSP^3 operators `and`, `or` and `not` to combine (parts of) constraints.
A. No. These operators cannot be redefined. For a predicate (expression), you must use |, & and ^; see Table 1.2. For posting two sets of constraints linked by `and`, simply post two separate lists.

Index

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] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog. Technical Report T2012:03, TASC-SICS-LINA, 2014.

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

[5] C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. SLIDE: A useful special case of the CARDPATH constraint. In *Proceedings of ECAI'08*, pages 475–479, 2008.

[6] 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.

[7] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio Link Frequency Assignment. *Constraints*, 4(1):79–89, 1999.

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

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

[10] M. Carlsson and N. Beldiceanu. Revisiting the lexicographic ordering constraint. Technical Report T2002-17, Swedish Institute of Computer Science, 2002.

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

[12] M. Carlsson, M. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proceedings of PLILP'97*, pages 191–306, 1997.

[13] K. Cheng and R. Yap. Maintaining generalized arc consistency on ad-hoc n-ary Boolean constraints. In *Proceedings of ECAI'06*, pages 78–82, 2006.

[14] K. Cheng and R. Yap. Maintaining generalized arc consistency on ad-hoc r-ary constraints. In *Proceedings of CP'08*, pages 509–523, 2008.

[15] Kenil C. K. Cheng and Roland H. C. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[16] R. Cymer. Dulmage-mendelsohn canonical decomposition as a generic pruning technique. *Constraints*, 17(3):234–272, 2012.

[17] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proceedings of CP'02*, pages 93–108, 2002.

[18] I.P. Gent, I. Miguel, and P. Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence*, 172(18):1973–2000, 2008.

[19] 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.

[20] 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.

[21] 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.

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

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

[24] C. Lecoutre. *Constraint networks: techniques and algorithms*. ISTE/Wiley, 2009.

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

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

[27] Guillaume Perez and Jean-Charles Régin. Improving GAC-4 for Table and MDD constraints. In *Proceedings of CP'14*, pages 606–621, 2014.

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

[29] G. Pesant and C.-G. Quimper. Counting solutions of knapsack constraints. In *Proceedings of CPAIOR'08*, pages 203–217, 2008.

[30] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, pages 362–367, 1994.

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

[32] M. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals OR*, 118(1-4):73–84, 2003.

[33] W.J. van Hoeve. The alldifferent constraint: a survey. In *Proceedings of the Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.

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

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