Constraint Programming
– Modeling –

Christophe Lecoutre

CRIL-CNRS UMR 8188
Universite d'Artois
Lens, France

January 2021

Outline

Outline

CP = Modeling + Solving

1. Modeling: describing the real-world problem in a declarative way, typically as :
   - a CSP (Constraint Satisfaction Problem) instance, or
   - a COP (Constraint Optimization Problem) instance.
2. Solving: applying efficient techniques to explore the search space, in order to find solutions.

Modeling and Solving are somehow independent processes:

- a model can be applied various search/solving techniques
- a solver can be applied to various problems/models

CP = Modeling + Solving

1. Modeling: describing the real-world problem in a declarative way, typically as :
   - a CSP (Constraint Satisfaction Problem) instance, or
   - a COP (Constraint Optimization Problem) instance.
2. Solving: applying efficient techniques to explore the search space, in order to find solutions.

Modeling and Solving are somehow independent processes:

- a model can be applied various search/solving techniques
- a solver can be applied to various problems/models

CP = Modeling + Solving

1. Modeling: describing the real-world problem in a declarative way, typically as :
   - a CSP (Constraint Satisfaction Problem) instance, or
   - a COP (Constraint Optimization Problem) instance.
2. Solving: applying efficient techniques to explore the search space, in order to find solutions.

Modeling and Solving are somehow independent processes:

- a model can be applied various search/solving techniques
- a solver can be applied to various problems/models

CP = Modeling + Solving

1. Modeling: describing the real-world problem in a declarative way, typically as :
   - a CSP (Constraint Satisfaction Problem) instance, or
   - a COP (Constraint Optimization Problem) instance.
2. Solving: applying efficient techniques to explore the search space, in order to find solutions.

Modeling and Solving are somehow independent processes:

- a model can be applied various search/solving techniques
- a solver can be applied to various problems/models

CP = Modeling + Solving

1. Modeling: describing the real-world problem in a declarative way, typically as :
   - a CSP (Constraint Satisfaction Problem) instance, or
   - a COP (Constraint Optimization Problem) instance.
2. Solving: applying efficient techniques to explore the search space, in order to find solutions.

Modeling and Solving are somehow independent processes:

- a model can be applied various search/solving techniques
- a solver can be applied to various problems/models

CP = Modeling + Solving

1. Modeling: describing the real-world problem in a declarative way, typically as :
   - a CSP (Constraint Satisfaction Problem) instance, or
   - a COP (Constraint Optimization Problem) instance.
2. Solving: applying efficient techniques to explore the search space, in order to find solutions.

Modeling and Solving are somehow independent processes:

- a model can be applied various search/solving techniques
- a solver can be applied to various problems/models

CSP/COP Instances

Definition

An instance P of the Constraint Satisfaction Problem (CSP), also called a Constraint Network (CN), is composed of:

- a finite set of variables, denoted by $vars(P)$,
- a finite set of constraints, denoted by $ctrs(P)$.

Definition

An instance P of the Constraint Optimisation Problem (COP) additionally involves:

- an objective function $obj(P)$ to be minimized or maximized.

CSP/COP Instances

Definition

An instance P of the Constraint Satisfaction Problem (CSP), also called a Constraint Network (CN), is composed of:

- a finite set of variables, denoted by $vars(P)$,
- a finite set of constraints, denoted by $ctrs(P)$.

Definition

An instance P of the Constraint Optimisation Problem (COP) additionally involves:

- an objective function $obj(P)$ to be minimized or maximized.

Declarativity

Modeling is a declarative process.

> *"Describe what you see/want. Not, how to get it!"*

A CSP/COP instance describes what the solutions are like, and specifies the search space:

$$dom(x_1) \times dom(x_2) \times \cdots \times dom(x_n)$$

Variables represents the view we have of the problem. There are several types of variables:

- Boolean variables
- integer variables
- real variables
- set variables
- ...

Declarativity

Modeling is a declarative process.

> *"Describe what you see/want. Not, how to get it!"*

A CSP/COP instance describes what the solutions are like, and specifies the search space:

$$dom(x_1) \times dom(x_2) \times \cdots \times dom(x_n)$$

Variables represents the view we have of the problem. There are several types of variables:

- Boolean variables
- integer variables
- real variables
- set variables
- ...

Declarativity

Modeling is a declarative process.

> *"Describe what you see/want. Not, how to get it!"*

A CSP/COP instance describes what the solutions are like, and specifies the search space:

$$dom(x_1) \times dom(x_2) \times \cdots \times dom(x_n)$$

Variables represents the view we have of the problem. There are several types of variables:

- Boolean variables
- integer variables
- real variables
- set variables
- ...

Declarativity

Modeling is a declarative process.

"Describe what you see/want. Not, how to get it!"

A CSP/COP instance describes what the solutions are like, and specifies the search space:

$$dom(x_1) \times dom(x_2) \times \cdots \times dom(x_n)$$

Variables represents the view we have of the problem. There are several types of variables:

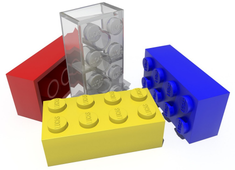
- Boolean variables
- integer variables our focus!
- real variables
- set variables
- ...

Constraints

Basically, CP is thinking Constraints.

Constraints are building blocks of constraint reasoning:

- used to model the problem
- used to solve the problem



Remember that a constraint c has:

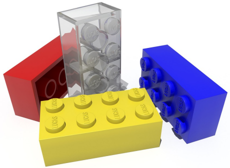
- a scope $scp(c)$: the variables involved in c
- a relation $rel(c)$: the combinations of values accepted by c

Constraints

Basically, CP is thinking Constraints.

Constraints are building blocks of constraint reasoning:

- used to model the problem
- used to solve the problem



Remember that a constraint c has:

- a scope $scp(c)$: the variables involved in c
- a relation $rel(c)$: the combinations of values accepted by c

Outline

1. Reminder
2. Languages and Formats
3. Some Popular Constraints
   - Generic Constraints
   - Case Study "Nonogram" introducing `regular`
   - Case Study "Sudoku" introducing `allDifferent`
   - Case Study "Magic Sequence" introducing `sum` and `cardinality`
   - Case Study "Warehouse Location" introducing `count` and `element`
   - Case Study "Black hole" introducing `channel`

Modeling Languages

Modeling languages can be used to represent problems, using some form of control and abstraction.

Typically, a model captures a family of problem instances, by referring to some parameters representing the data. Building a model for a problem involves:

1. identifying the **parameters**, i.e., the structure of the data
2. writting the **model**, by taking the parameters into account, and using an appropriate (high-level) language

Once we have a model, we still have to provide the effective **data** for each specific instance to be treated.

Modeling Languages

Modeling languages can be used to represent problems, using some form of control and abstraction.

Typically, a model captures a family of problem instances, by referring to some parameters representing the data. Building a model for a problem involves:

1. identifying the **parameters**, i.e., the structure of the data
2. writting the **model**, by taking the parameters into account, and using an appropriate (high-level) language

Once we have a model, we still have to provide the effective **data** for each specific instance to be treated.

Modeling Languages

Modeling languages can be used to represent problems, using some form of control and abstraction.

Typically, a model captures a family of problem instances, by referring to some parameters representing the data. Building a model for a problem involves:

1. identifying the **parameters**, i.e., the structure of the data
2. writting the **model**, by taking the parameters into account, and using an appropriate (high-level) language

Once we have a model, we still have to provide the effective **data** for each specific instance to be treated.

Modeling Languages

Modeling languages can be used to represent problems, using some form of control and abstraction.

Typically, a model captures a family of problem instances, by referring to some parameters representing the data. Building a model for a problem involves:

1. identifying the **parameters**, i.e., the structure of the data
2. writting the **model**, by taking the parameters into account, and using an appropriate (high-level) language

Once we have a model, we still have to provide the effective **data** for each specific instance to be treated.

Let us illustrate this with the academic problem "`All-Interval Series`"

All-Interval Series (CSPLib 007)

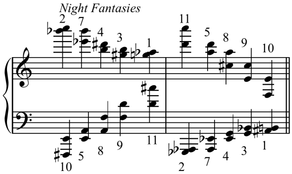
Given an integer n, find a sequence $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ such that:

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

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

1 7 0 5 4 2 6 3

All-Interval Series (CSPLib 007)



Given an 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. $y = \langle |x_1 - x_0|, |x_2 - x_1|, ..., |x_{n-1} - x_{n-2}| \rangle$ is a permutation of $\{1, 2, ..., n-1\}$

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

1 7 0 5 4 2 6 3

All-Interval Series (CSPLib 007)

Given an 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. $y = \langle |x_1 - x_0|, |x_2 - x_1|, ..., |x_{n-1} - x_{n-2}| \rangle$ is a permutation of $\{1, 2, ..., n-1\}$

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

1 7 0 5 4 2 6 3

All-Interval Series (CSPLib 007)

Given an 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. $y = \langle |x_1 - x_0|, |x_2 - x_1|, ..., |x_{n-1} - x_{n-2}| \rangle$ is a permutation of $\{1, 2, ..., n-1\}$

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

1 7 0 5 4 2 6 3

Data for All-Interval Series

First, we have to identify the **parameters** (structure of the data).

Here, we just need an integer for representing the order (n) of the problem instance.

Which format for representing effective data?

- Tabular (Text)? / XML? / JSON?

JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{ "n": 5 }
```

Remark.
Technically, when the data are very basic, there is no real need to generate such data files.

Data for `All-Interval Series`

First, we have to identify the **parameters** (structure of the data).

Here, we just need an integer for representing the order (n) of the problem instance.

Which format for representing effective data?

- Tabular (Text)? / XML? / JSON?

JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{ "n": 5 }
```

Remark.
Technically, when the data are very basic, there is no real need to generate such data files.

Data for `All-Interval Series`

First, we have to identify the **parameters** (structure of the data).

Here, we just need an integer for representing the order (n) of the problem instance.

Which format for representing effective data?

- Tabular (Text)? / XML? / JSON?

JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{ "n": 5 }
```

Remark.
Technically, when the data are very basic, there is no real need to generate such data files.

Data for `All-Interval Series`

First, we have to identify the **parameters** (structure of the data).

Here, we just need an integer for representing the order (n) of the problem instance.

Which format for representing effective data?

- Tabular (Text)? / XML? / JSON?

JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{ "n": 5 }
```

Remark.
Technically, when the data are very basic, there is no real need to generate such data files.

Data for `All-Interval Series`

First, we have to identify the **parameters** (structure of the data).

Here, we just need an integer for representing the order (n) of the problem instance.

Which format for representing effective data?

- Tabular (Text)? / XML? / JSON?

JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{ "n": 5 }
```

Remark.

Technically, when the data are very basic, there is no real need to generate such data files.

Data for `All-Interval Series`

First, we have to identify the **parameters** (structure of the data).

Here, we just need an integer for representing the order (n) of the problem instance.

Which format for representing effective data?

- Tabular (Text)? / XML? / JSON?

JSON is a good choice for representing effective data. For example, for order 5, we can generate a file containing:

```
{ "n": 5 }
```

a lightweight data-interchange format (our choice)

Remark.

Technically, when the data are very basic, there is no real need to generate such data files.

Model for `All-Interval Series`

Second, we have to write the **model**.

With n being the unique "parameter" of this problem, the structure of a natural model is:

- Variables
  - x, one-dimensional array of n integer variables
- Constraints
  - a constraint `allDifferent` on x
  - a constraint `allDifferent` on "y"

Which language to choose for writting models?

- AMPL? / OPL? / MiniZinc? / Essence?
- PyCSP^3

Model for `All-Interval Series`

Second, we have to write the **model**.

With n being the unique "parameter" of this problem, the structure of a natural model is:

- Variables
  - x, one-dimensional array of n integer variables
- Constraints
  - a constraint `allDifferent` on x
  - a constraint `allDifferent` on "y"

Which language to choose for writting models?

- AMPL? / OPL? / MiniZinc? / Essence?
- PyCSP^3

Model for `All-Interval Series`

Second, we have to write the **model**.

With n being the unique "parameter" of this problem, the structure of a natural model is:

- Variables
  - x, one-dimensional array of n integer variables
- Constraints
  - a constraint `allDifferent` on x
  - a constraint `allDifferent` on "y"

Which language to choose for writting models?

- AMPL? / OPL? / MiniZinc? / Essence?
- PyCSP^3

Model for `All-Interval Series`

Second, we have to write the **model**.

With n being the unique "parameter" of this problem, the structure of a natural model is:

- Variables
  - x, one-dimensional array of n integer variables
- Constraints
  - a constraint `allDifferent` on x
  - a constraint `allDifferent` on "y"

Which language to choose for writting models?

- AMPL? / OPL? / MiniZinc? / Essence?
- PyCSP^3 a Python Library (our choice)

PyCSP^3 Model for `All-Interval Series`

File `AllInterval.py`

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

Modelling Languages and Solvers

Unfortunately, most of the solvers cannot directly read/understand modeling languages. For each problem instance, identified by a model and effective data, we have to generate a specific representation (new file).

Which format to choose for representing separate instances?

- XCSP 2.1
- FlatZinc
- XCSP^3

Important:

- XCSP 2.1 and FlatZinc are flat formats
- XCSP^3 is an intermediate format preserving the model structure

Modelling Languages and Solvers

Unfortunately, most of the solvers cannot directly read/understand modeling languages. For each problem instance, identified by a model and effective data, we have to generate a specific representation (new file).

Which format to choose for representing separate instances?

- XCSP 2.1
- FlatZinc
- XCSP^3

Important:

- XCSP 2.1 and FlatZinc are flat formats
- XCSP^3 is an intermediate format preserving the model structure

Modelling Languages and Solvers

Unfortunately, most of the solvers cannot directly read/understand modeling languages. For each problem instance, identified by a model and effective data, we have to generate a specific representation (new file).

Which format to choose for representing separate instances?

- XCSP 2.1
- FlatZinc
- XCSP^3

Important:

- XCSP 2.1 and FlatZinc are flat formats
- XCSP^3 is an intermediate format preserving the model structure

Modelling Languages and Solvers

Unfortunately, most of the solvers cannot directly read/understand modeling languages. For each problem instance, identified by a model and effective data, we have to generate a specific representation (new file).

Which format to choose for representing separate instances?

- XCSP 2.1
- FlatZinc
- XCSP^3 an XML-based representation (our choice)

Important:

- XCSP 2.1 and FlatZinc are flat formats
- XCSP^3 is an intermediate format preserving the model structure

XCSP3 Instance: AllInterval-05

Third, we have to provide the effective **data**.

```
python3 AllInterval.py -data=5
```

We obtain:

File `AllInterval-5.xml`

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i] is the ith series note" size="[5]">
      0..4
    </array>
  </variables>
  <constraints>
    <allDifferent note="notes must occur once...">
      x[]
    </allDifferent>
    <allDifferent note="intervals between neighbouring notes ...">
      dist(x[1],x[0]) dist(x[2],x[1]) dist(x[3],x[2]) dist(x[4],x[3])
    </allDifferent>
  </constraints>
</instance>
```

XCSP3 Instance: AllInterval-05

Third, we have to provide the effective **data**.

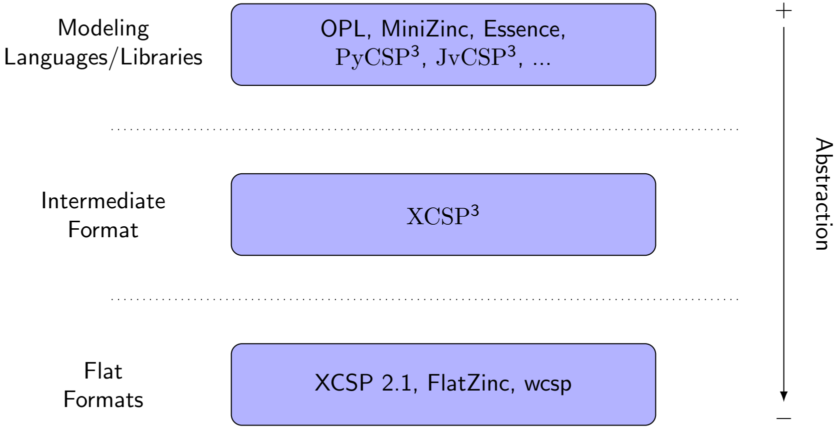
```
python3 AllInterval.py -data=5
```

We obtain:

File `AllInterval-5.xml`

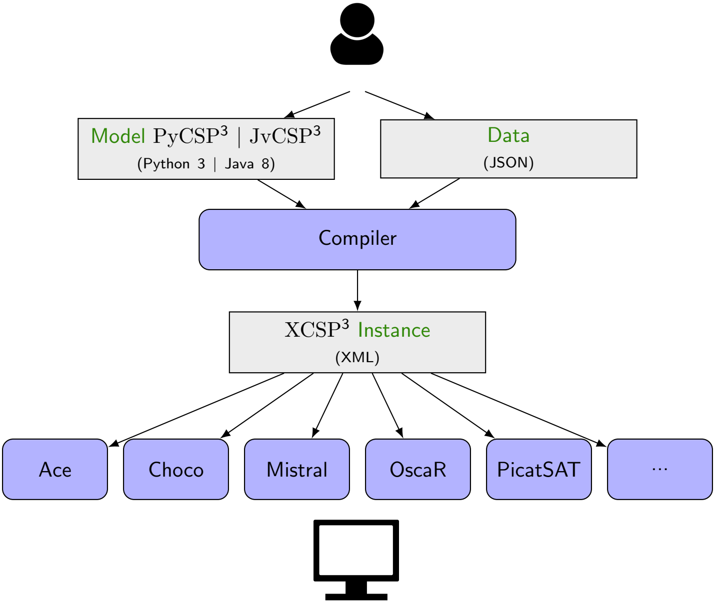
```xml
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i] is the ith series note" size="[5]">
      0..4
    </array>
  </variables>
  <constraints>
    <allDifferent note="notes must occur once...">
      x[]
    </allDifferent>
    <allDifferent note="intervals between neighbouring notes ...">
      dist(x[1],x[0]) dist(x[2],x[1]) dist(x[3],x[2]) dist(x[4],x[3])
    </allDifferent>
  </constraints>
</instance>
```

Modeling Languages and Formats



www.xcsp.org

A Complete Modeling/Solving Toolchain



Mainstream Technologies

The complete Toolchain $\mathrm{PyCSP}^3 + \mathrm{XCSP}^3$ has many advantages:

- Python (and Java), JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 (or Java 8) avoids the user learning a new programming language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.
At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain $\mathrm{PyCSP}^3 + \mathrm{XCSP}^3$ has many advantages:

- Python (and Java), JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 (or Java 8) avoids the user learning a new programming language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.
At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain $\mathrm{PyCSP}^3 + \mathrm{XCSP}^3$ has many advantages:

- Python (and Java), JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 (or Java 8) avoids the user learning a new programming language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.
At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain $\mathrm{PyCSP}^3 + \mathrm{XCSP}^3$ has many advantages:

- Python (and Java), JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 (or Java 8) avoids the user learning a new programming language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.
At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain $\mathrm{PyCSP}^3 + \mathrm{XCSP}^3$ has many advantages:

- Python (and Java), JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 (or Java 8) avoids the user learning a new programming language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.
At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain $\mathrm{PyCSP}^3 + \mathrm{XCSP}^3$ has many advantages:

- Python (and Java), JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 (or Java 8) avoids the user learning a new programming language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.

At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Did we choose the Right Language?

PHP?

- Is that true that any variable identifier must start with \$?
- Why not using €?
- Is that true that we can find method identifiers such as:
  - similar_text
  - addslashes

  ?

Asking the question was a joke!

PHP?

- Is that true that any variable identifier must start with $?
- Why not using €?
- Is that true that we can find method identifiers such as:
  - similar_text
  - addslashes

  ?

Asking the question was a joke!

PHP?

- Is that true that any variable identifier must start with $?
- Why not using €?
- Is that true that we can find method identifiers such as:
  - similar_text
  - addslashes

  ?

Asking the question was a joke!

PHP?

- Is that true that any variable identifier must start with $?
- Why not using €?
- Is that true that we can find method identifiers such as:
  - similar_text
  - addslashes

  ?

Asking the question was a joke!

PHP?

- Is that true that any variable identifier must start with $?
- Why not using €?
- Is that true that we can find method identifiers such as:
  - similar_text
  - addslashes

  ?

Asking the question was a joke!

JavaScript?

Evaluate the following expressions:

```
"2" == 2
"2" === 2
[] == []
[] == ![]
2 == [2]
true + true
"toto" instanceof String
0.1 + 0.2 == 0.3
"2" + 1
"2" - 1
```

JavaScript?

Evaluate the following expressions:

```
"2" == 2
"2" === 2
[] == []
[] == ![]
2 == [2]
true + true
"toto" instanceof String
0.1 + 0.2 == 0.3
"2" + 1
"2" - 1
```

JavaScript?

Evaluate the following expressions:

`"2" == 2`	`true`
`"2" === 2`	`false`
`[] == []`	`false`
`[] == ![]`	`true`
`2 == [2]`	`true`
`true + true`	`2`
`"toto" instanceof String`	`false`
`0.1 + 0.2 == 0.3`	`false`
`"2" + 1`	`"21"`
`"2" - 1`	`1`

C++?

C++

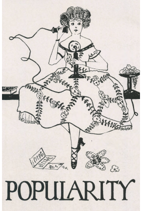
- Too hard to implement and to learn: the specification has grown to over 1000 pages.
- Everybody uses a different subset of the language, making it harder to understand others' code.

Four ways of declaring an array?

Outline

1. Reminder
2. Languages and Formats
3. Some Popular Constraints
   - Generic Constraints
   - Case Study "Nonogram" introducing `regular`
   - Case Study "Sudoku" introducing `allDifferent`
   - Case Study "Magic Sequence" introducing `sum` and `cardinality`
   - Case Study "Warehouse Location" introducing `count` and `element`
   - Case Study "Black hole" introducing `channel`

Popular Constraints

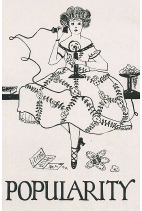


Popular constraints are those that are:

- often used when modeling problems
- implemented in many solvers

Remark.
XCSP^3-core contains popular constraints over integer variables, classified by families.

Popular Constraints

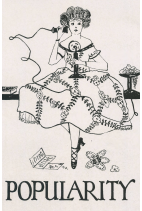


Popular constraints are those that are:

- often used when modeling problems
- implemented in many solvers

Remark.
XCSP^3-core contains popular constraints over integer variables, classified by families.

Popular Constraints



Popular constraints are those that are:

- often used when modeling problems
- implemented in many solvers

Remark.
XCSP^3-core contains popular constraints over integer variables, classified by families.

XCSP^3-core

Constraints over Integer Variables

- Generic Constraints
  - intension, extension
- Language-based Constraints
  - regular, mdd
- Comparison-based Constraints
  - allDifferent, allEqual
  - ordered, lex
- Counting and Summing Constraints
  - sum (linear)
  - count (capturing atLeast, atMost,exactly, among)
  - nValues, cardinality
- Connection Constraints
  - minimum, maximum
  - element, channel
- Packing and Scheduling Constraints
  - noOverlap (capturing disjunctive and diffn)
  - cumulative

XCSP^3-core

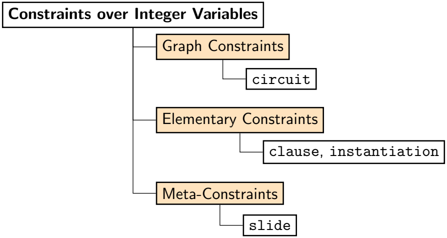
- **Constraints over Integer Variables**
  - Graph Constraints
    - `circuit`
  - Elementary Constraints
    - `clause`, `instantiation`
  - Meta-Constraints
    - `slide`

Note that XCSP^3-core is;

- sufficient for modeling many problems
- used in XCSP^3 Solver Competititons

Remark.
We shall introduce some of these constraints in disorder (guided by case studies).

XCSP^3-core



Note that XCSP^3-core is;

- sufficient for modeling many problems
- used in XCSP^3 Solver Competititons

Remark.
We shall introduce some of these constraints in disorder (guided by case studies).

XCSP³-core

Constraints over Integer Variables

- Graph Constraints
  - `circuit`
- Elementary Constraints
  - `clause`, `instantiation`
- Meta-Constraints
  - `slide`

Note that XCSP³-core is;

- sufficient for modeling many problems
- used in XCSP³ Solver Competititons

Remark.

We shall introduce some of these constraints in disorder (guided by case studies).

Outline

Any constraint given by a Boolean expression (predicate) built from:

- variables,
- constants (integers),
- arithmetic, relational, set and logical operators.

Example.

- $x > 2$
- $x \leq y + 1$
- $|x - y| = z - w$
- $x + y * 12 + z/2 = 5$
- $x + y > 3 \vee x * z = w$

Remark.

Above, the examples are given in "pure" mathematical forms. For PyCSP^3, operators are those of Python.

Any constraint given by a Boolean expression (predicate) built from:

- variables,
- constants (integers),
- arithmetic, relational, set and logical operators.

Example.

- $x > 2$
- $x \leq y + 1$
- $|x - y| = z - w$
- $x + y * 12 + z/2 = 5$
- $x + y > 3 \vee x * z = w$

Remark.

Above, the examples are given in "pure" mathematical forms. For PyCSP^3, operators are those of Python.

Any constraint given by a Boolean expression (predicate) built from:

- variables,
- constants (integers),
- arithmetic, relational, set and logical operators.

Example.

- $x > 2$
- $x \leq y + 1$
- $|x - y| = z - w$
- $x + y * 12 + z/2 = 5$
- $x + y > 3 \vee x * z = w$

Remark.
Above, the examples are given in "pure" mathematical forms. For PyCSP^3, operators are those of Python.

Operators used by PyCSP^3

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

Illustration

Mathematical forms:

- $x > 2$
- $x \leq y + 1$
- $|x - y| = z - w$
- $x + y * 12 + z/2 = 5$
- $x + y > 3 \vee x * z = w$

PyCSP^3 forms:

- $x > 2$
- $x <= y + 1$
- $abs(x - y) == z - w$
- $x + y * 12 + z//2 == 5$
- $(x + y > 3) \,|\, (x * z == w)$

When compiling from PyCSP^3 to XCSP^3, we obtain functional forms:

```
<intension> gt(x,2) </intension>
<intension> le(x,add(y,1)) </intension>
<intension> eq(dist(x,y),sub(z,w)) </intension>
<intension> eq(add(x,mul(y,12),div(z,2)),5) </intension>
<intension> or(gt(add(x,y),3),eq(mul(x,z),w)) </intension>
```

Illustration

Mathematical forms:

- $x > 2$
- $x \leq y + 1$
- $|x - y| = z - w$
- $x + y * 12 + z/2 = 5$
- $x + y > 3 \vee x * z = w$

PyCSP^3 forms:

- $x > 2$
- $x <= y + 1$
- $abs(x - y) == z - w$
- $x + y * 12 + z//2 == 5$
- $(x + y > 3) \,|\, (x * z == w)$

When compiling from PyCSP^3 to XCSP^3, we obtain functional forms:

```
<intension> gt(x,2) </intension>
<intension> le(x,add(y,1)) </intension>
<intension> eq(dist(x,y),sub(z,w)) </intension>
<intension> eq(add(x,mul(y,12),div(z,2)),5) </intension>
<intension> or(gt(add(x,y),3),eq(mul(x,z),w)) </intension>
```

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a positive table constraint,
- $X \notin T$ is a negative table constraint.

Remark.
Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (Hot research topic); Not in XCSP^3-core

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a positive table constraint,
- $X \notin T$ is a negative table constraint.

Remark.
Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (Hot research topic); Not in XCSP^3-core

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a positive table constraint,
- $X \notin T$ is a negative table constraint.

Remark.

Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (Hot research topic); Not in XCSP^3-core

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a positive table constraint,
- $X \notin T$ is a negative table constraint.

Remark.
Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (Hot research topic); Not in XCSP^3-core

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a positive table constraint,
- $X \notin T$ is a negative table constraint.

Remark.
Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (Hot research topic); Not in XCSP^3-core

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a positive table constraint,
- $X \notin T$ is a negative table constraint.

Remark.

Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (Hot research topic); Not in XCSP^3-core

Generic Constraint `extension`

The table constraint:

x	y	z
0	0	0
0	0	1
0	0	2
1	1	1
1	2	2
2	2	0

is written in PyCSP^3 as:

```
(x,y,z) in {(0,0,0), (0,0,1), (0,0,2), (1,1,1), (1,2,2), (2,2,0)}
```

If the domain of the variable z is $\{0, 1, 2\}$, can we compress?

x	y	z
0	0	$*$
1	1	1
1	2	2
2	2	0

which gives in PyCSP^3:

```
(x,y,z) in {(0,0,ANY), (1,1,1), (1,2,2), (2,2,0)}
```

and gives in XCSP^3:

```
<extension>
  <list> x y z </list>
  <supports> (0,0,*)(1,1,1)(1,2,2)(2,2,0) </supports>
<extension>
```

Generic Constraint `extension`

If the domain of the variable z is $\{0, 1, 2\}$, can we compress?

x	y	z
0	0	$*$
1	1	1
1	2	2
2	2	0

which gives in PyCSP^3:

```
(x,y,z) in {(0,0,ANY), (1,1,1), (1,2,2), (2,2,0)}
```

and gives in XCSP^3:

```
<extension>
  <list> x y z </list>
  <supports> (0,0,*)(1,1,1)(1,2,2)(2,2,0) </supports>
<extension>
```

If the domain of the variable z is $\{0, 1, 2\}$, can we compress?

x	y	z
0	0	$*$
1	1	1
1	2	2
2	2	0

which gives in PyCSP^3:

```
(x,y,z) in {(0,0,ANY), (1,1,1), (1,2,2), (2,2,0)}
```

and gives in XCSP^3:

```
<extension>
  <list> x y z </list>
  <supports> (0,0,*)(1,1,1)(1,2,2)(2,2,0) </supports>
<extension>
```

If the domain of the variable z is $\{0, 1, 2\}$, can we compress?

x	y	z
0	0	$*$
1	1	1
1	2	2
2	2	0

which gives in PyCSP^3:

```
(x,y,z) in {(0,0,ANY), (1,1,1), (1,2,2), (2,2,0)}
```

and gives in XCSP^3:

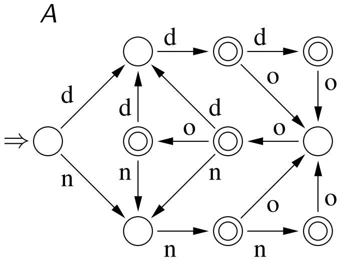
```
<extension>
  <list> x y z </list>
  <supports> (0,0,*)(1,1,1)(1,2,2)(2,2,0) </supports>
<extension>
```

Outline

With X a sequence of variables and A a deterministic (or non-deterministic) finite automaton, $X \in A$ is a constraint `regular`.

Example.

A constraint `regular`
$\langle x_1, x_2, x_3, x_4, x_5 \rangle \in A$



Remark.

In PyCSP^3, for posting a `regular` constraint, we use the Python operator 'in' (as for table constraints) and a PyCSP^3 object called `Automaton`.

Remark.

An instantiation of X satisfies the constraint if it represents a word recognized by the automaton A.

Example.

For the previous constraint:

- $\{x_1 = d, x_2 = d, x_3 = d, x_4 = o, x_5 = o\}$ satisfies the constraint
- $\{x_1 = d, x_2 = d, x_3 = o, x_4 = o, x_5 = n\}$ does not satisfy the constraint

Remark.

It is possible de convert a constraint `regular` into:

- a constraint `extension` (but possible memory space explosion)
- a related constraint called `mdd` (performed in solver Ace)

Remark.

An instantiation of X satisfies the constraint if it represents a word recognized by the automaton A.

Example.

For the previous constraint:

- $\{x_1 = d, x_2 = d, x_3 = d, x_4 = o, x_5 = o\}$ satisfies the constraint
- $\{x_1 = d, x_2 = d, x_3 = o, x_4 = o, x_5 = n\}$ does not satisfy the constraint

Remark.

It is possible de convert a constraint `regular` into:

- a constraint `extension` (but possible memory space explosion)
- a related constraint called `mdd` (performed in solver Ace)

Remark.

An instantiation of X satisfies the constraint if it represents a word recognized by the automaton A.

Example.

For the previous constraint:

- $\{x_1 = d, x_2 = d, x_3 = d, x_4 = o, x_5 = o\}$ satisfies the constraint
- $\{x_1 = d, x_2 = d, x_3 = o, x_4 = o, x_5 = n\}$ does not satisfy the constraint

Remark.

It is possible de convert a constraint `regular` into:

- a constraint `extension` (but possible memory space explosion)
- a related constraint called `mdd` (performed in solver Ace)

Nonogram Puzzle

				2	2	2	2	2	2	2	
			3	3	2	2	2	2	2	3	3
	2	2									
	4	4									
1	3	1									
2	1	2									
	1	1									
	2	2									
	2	2									
		3									
		1									

Solution to the Nonogram Puzzle

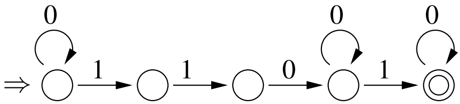
				2	2	2	2	2	2	2	
			3	3	2	2	2	2	2	3	3
	2	2		■	■				■	■	
	4	4	■	■	■	■		■	■	■	■
1	3	1	■			■	■	■			■
2	1	2	■	■			■			■	■
	1	1		■						■	
	2	2		■	■				■	■	
	2	2			■	■		■	■		
		3				■	■	■			
		1					■				

Using `regular` for Nonogram

Remark.
Each clue corresponds to a regular expression

Example.
The clue 2 1 corresponds to:

$0^*1^20^+10^*$



When we consider the benchmark proposed by G. Pesant:

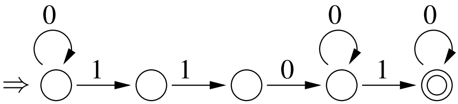
- tables are huge (more than $1,000,000$ tuples for some of them)
- MDDs are rather compact (a few hundreds of nodes, at the most)

Using `regular` for Nonogram

Remark.
Each clue corresponds to a regular expression

Example.
The clue 2 1 corresponds to:

$0^*1^20^+10^*$



When we consider the benchmark proposed by G. Pesant:

- tables are huge (more than $1,000,000$ tuples for some of them)
- MDDs are rather compact (a few hundreds of nodes, at the most)

Specifying Data

The data for the previous Nonogram puzzle can simply be in JSON:

```
{
  "rowPatterns":
    [[2,2],[4,4],[1,3,1],[2,1,2],[1,1],[2,2],[2,2],[3],[1]],
  "colPatterns":
    [[3],[2,3],[2,2],[2,2],[2,2],[2,2],[2,2],[2,3],[3]]
}
```

Remark.

Remember that we store the data corresponding to each instance in a specific file (here, called 'heart.json').

Specifying Data

The data for the previous Nonogram puzzle can simply be in JSON:

```
{
  "rowPatterns":
    [[2,2],[4,4],[1,3,1],[2,1,2],[1,1],[2,2],[2,2],[3],[1]],
  "colPatterns":
    [[3],[2,3],[2,2],[2,2],[2,2],[2,2],[2,2],[2,3],[3]]
}
```

Remark.

Remember that we store the data corresponding to each instance in a specific file (here, called 'heart.json').

PyCSP^3 Model

For the model, we use tuple unpacking, and NumPy-like notations:

File Nonogram.py

```
from pycsp3 import *

row_patterns, col_patterns = data
nRows, nCols = len(row_patterns), len(col_patterns)

# x[i][j] is 1 iff the cell at row i and col j is colored in black
x = VarArray(size=[nRows, nCols], dom={0, 1})

def automaton(pattern):
  ...  # to be written

satisfy(
  [x[i] in automaton(row_patterns[i]) for i in range(nRows)],

  [x[:, j] in automaton(col_patterns[j]) for j in range(nCols)]
)
```

```
python3 Nonogram.py -data=heart.json
```

Outline

Semantics

`allDifferent`(X, E), `with` $X = \langle x_1, x_2, \ldots \rangle$, `iff`
$\forall (i,j) : 1 \leq i < j \leq |X|, \boldsymbol{x}_i \neq \boldsymbol{x}_j \vee \boldsymbol{x}_i \in E \vee \boldsymbol{x}_j \in E$
`allDifferent`(X) `iff` `allDifferent`$(X, \emptyset)$

Semantics

`allDifferent-matrix`$(\mathcal{M})$, `with` $\mathcal{M}$ `a matrix of variables of size` $n \times m$, `iff`
$\forall i : 1 \leq i \leq n,$ `allDifferent`$(\mathcal{M}[i])$
$\forall j : 1 \leq j \leq m,$ `allDifferent`$(\mathcal{M}^T[j])$

Remark.
One form accepts excepting values, and another is lifted to matrices.

Remark.
In PyCSP^3, we call the function `AllDifferent()` that accepts two optional named parameters called `excepting` and `matrix`.

Global Constraint `allDifferent`

Semantics

`allDifferent`(X, E), `with` $X = \langle x_1, x_2, \ldots \rangle$, `iff`
$\forall (i,j) : 1 \leq i < j \leq |X|, \boldsymbol{x_i} \neq \boldsymbol{x_j} \vee \boldsymbol{x_i} \in E \vee \boldsymbol{x_j} \in E$
`allDifferent`(X) `iff` `allDifferent`$(X, \emptyset)$

Semantics

`allDifferent-matrix`$(\mathcal{M})$, `with` $\mathcal{M}$ `a matrix of variables of size` $n \times m$, `iff`
$\forall i : 1 \leq i \leq n,$ `allDifferent`$(\mathcal{M}[i])$
$\forall j : 1 \leq j \leq m,$ `allDifferent`$(\mathcal{M}^T[j])$

Remark.

One form accepts excepting values, and another is lifted to matrices.

Remark.

In PyCSP^3, we call the function `AllDifferent()` that accepts two optional named parameters called `excepting` and `matrix`.

Semantics

`allDifferent`(X, E), with $X = \langle x_1, x_2, \ldots \rangle$, iff
$\forall(i,j) : 1 \leq i < j \leq |X|, \boldsymbol{x}_i \neq \boldsymbol{x}_j \vee \boldsymbol{x}_i \in E \vee \boldsymbol{x}_j \in E$
`allDifferent`(X) iff `allDifferent`$(X, \emptyset)$

Semantics

`allDifferent-matrix`$(\mathcal{M})$, with $\mathcal{M}$ a matrix of variables of size $n \times m$, iff
$\forall i : 1 \leq i \leq n,$ `allDifferent`$(\mathcal{M}[i])$
$\forall j : 1 \leq j \leq m,$ `allDifferent`$(\mathcal{M}^T[j])$

Remark.

One form accepts excepting values, and another is lifted to matrices.

Remark.

In PyCSP^3, we call the function `AllDifferent()` that accepts two optional named parameters called `excepting` and `matrix`.

Sudoku

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

The data (clues) must be stored in a JSON file; here a file grid.json:

```
{
  "clues": [[0,2,0,5,0,1,0,9,0],...,[0,1,0,9,0,7,0,6,0]]
}
```

Sudoku

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

The data (clues) must be stored in a JSON file; here a file grid.json:

```
{
  "clues": [[0,2,0,5,0,1,0,9,0],...,[0,1,0,9,0,7,0,6,0]]
}
```

PyCSP^3 Model

File Sudoku.py

```
from pycsp3 import *

clues = data

# x[i][j] is the value in cell 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[i][j] > 0]
)
```

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

File Sudoku-grid.xml

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

Outline

1. Reminder
2. Languages and Formats
3. Some Popular Constraints
   - Generic Constraints
   - Case Study "Nonogram" introducing `regular`
   - Case Study "Sudoku" introducing `allDifferent`
   - Case Study "Magic Sequence" introducing `sum` and `cardinality`
   - Case Study "Warehouse Location" introducing `count` and `element`
   - Case Study "Black hole" introducing `channel`

A constraint `sum` is a constraint of the form:

$\sum_{i=1}^{r} c_i x_i$ `<op>` L

where:

- $c_i \in \mathbb{Z}, \forall i \in 1..r$
- `<op>` $\in \{<, \leq, \geq, >, =, \neq, \in, \notin\}$
- L is an integer, a variable or an interval

Coefficients can also be given under the form of variables.

Remark.
In PyCSP^3, we must call the function `Sum()` (or use a dot product).

For the semantics, V is a sequence of values and O is assumed to be a sequence of variables (for simplicity).

Semantics

$\texttt{cardinality}(X, V, O)$, with $X = \langle x_1, x_2, \ldots \rangle$, $V = \langle v_1, v_2, \ldots \rangle$, $O = \langle o_1, o_2, \ldots \rangle$,
iff $\forall j : 1 \leq j \leq |V|, |\{i : 1 \leq i \leq |X| \wedge \boldsymbol{x}_i = v_j\}| = \boldsymbol{o}_j$

Remark.

In PyCSP^3, we must call the function `Cardinality()` that accepts a list of variables as first parameter, and a named parameter called `occurrences` whose value must be a dictionary.

Example.

We give an example where O contains intervals.

$$\texttt{cardinality}(\langle x, y, z\rangle, \{N, D, O\}, \{0..1, 1..1, 1..2\})$$

As an illustration, we have:

- Instantiation (N,D,O)
- Instantiation (O,D,O)
- Instantiation (D,D,O)

Example.

We give an example where O contains intervals.

$$\texttt{cardinality}(\langle x, y, z\rangle, \{N, D, O\}, \{0..1, 1..1, 1..2\})$$

As an illustration, we have:

- Instantiation (N,D,O) $\Rightarrow$ OK
- Instantiation (O,D,O)
- Instantiation (D,D,O)

Example.

We give an example where O contains intervals.

$$\texttt{cardinality}(\langle x, y, z\rangle, \{N, D, O\}, \{0..1, 1..1, 1..2\})$$

As an illustration, we have:

- Instantiation (N,D,O)
- Instantiation (O,D,O) $\Rightarrow$ OK
- Instantiation (D,D,O)

Example.

We give an example where O contains intervals.

$$\texttt{cardinality}(\langle x, y, z\rangle, \{N, D, O\}, \{0..1, 1..1, 1..2\})$$

As an illustration, we have:

- Instantiation (N,D,O)
- Instantiation (O,D,O)
- Instantiation (D,D,O) $\Rightarrow$ KO

Example.

We give an example where O contains intervals.

$$\texttt{cardinality}(\langle x, y, z\rangle, \{N, D, O\}, \{0..1, 1..1, 1..2\})$$

As an illustration, we have:

- Instantiation (N,D,O) $\Rightarrow$ OK
- Instantiation (O,D,O) $\Rightarrow$ OK
- Instantiation (D,D,O) $\Rightarrow$ KO

Magic Sequence

Problem 019, proposed by T. Walsh, on CSPLib.

"A magic sequence of length (order) n is a sequence of integers $v_0, v_1, \ldots, v_{n-1}$ between 0 and $n-1$, such that for each value $i \in 0..n-1$ the value i occurs exactly v_i times in the sequence."

For instance,

6 2 1 0 0 0 1 0 0 0

is a magic sequence of length 10 since:

- 0 occurs 6 times in it,
- 1 occurs twice,
- 2 occurs once,
- ...

Magic Sequence

Problem 019, proposed by T. Walsh, on CSPLib.

"A magic sequence of length (order) n is a sequence of integers $v_0, v_1, \ldots, v_{n-1}$ between 0 and $n-1$, such that for each value $i \in 0..n-1$ the value i occurs exactly v_i times in the sequence."

For instance,

6 2 1 0 0 0 1 0 0 0

is a magic sequence of length 10 since:

- 0 occurs 6 times in it,
- 1 occurs twice,
- 2 occurs once,
- ...

PyCSP^3 Model

File MagicSequence.py

```
from pycsp3 import *

n = data

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

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

  # tag(redundant-constraints)
  [Sum(x) == n, Sum((i - 1) * x[i] for i in range(n)) == 0]
)
```

```
python3 MagicSequence.py -data=10
```

Outline

Can you say with your words what is the semantics of this constraint?

Semantics

`count`$(X, V) \odot k$, `with` $X = \langle x_1, x_2, \ldots \rangle$, `iff`
$|\{i : 1 \leq i \leq |X| \wedge \boldsymbol{x}_i \in V\}| \odot \boldsymbol{k}$

Special cases of `count` are:

- `atLeast`
- `atMost`
- `exactly`
- `among`

Remark.

In PyCSP^3, we must call the function `Count()` that accepts a list of variables as first parameter, and a named parameter which is either `value` or `values`.

Can you say with your words what is the semantics of this constraint?

Semantics

`count`$(X, V) \odot k$, `with` $X = \langle x_1, x_2, \ldots \rangle$, `iff`
$|\{i : 1 \leq i \leq |X| \wedge \boldsymbol{x}_i \in V\}| \odot \boldsymbol{k}$

Special cases of `count` are:

- `atLeast`
- `atMost`
- `exactly`
- `among`

Remark.
In PyCSP^3, we must call the function `Count()` that accepts a list of variables as first parameter, and a named parameter which is either `value` or `values`.

Global Constraint `count`

Can you say with your words what is the semantics of this constraint?

Semantics

$\texttt{count}(X, V) \odot k$, with $X = \langle x_1, x_2, \ldots \rangle$, iff
$|\{i : 1 \leq i \leq |X| \wedge \boldsymbol{x}_i \in V\}| \odot \boldsymbol{k}$

Special cases of `count` are:

- `atLeast`
- `atMost`
- `exactly`
- `among`

Remark.

In PyCSP^3, we must call the function `Count()` that accepts a list of variables as first parameter, and a named parameter which is either `value` or `values`.

Global Constraint `element`

Semantics

`element`(X, v), `with` $X = \langle x_1, x_2, \ldots \rangle$, `iff` `// indexing assumed to start at 1`

$\exists i : 1 \leq i \leq |X| \land \boldsymbol{x_i} = \boldsymbol{v}$

`element`(X, i, v), `with` $X = \langle x_1, x_2, \ldots \rangle$, `iff`

$\boldsymbol{x_i} = \boldsymbol{v}$

- The first form of constraint `element` allows us to test the membership of an element in a list.
- The second form allows us to make a connection between a list of variables (or integers) and a variable; this is the usual case.

Remark.
In PyCSP^3, we use natural indexing on lists (see Problem Warehouse).

Semantics

`element`(X, v), `with` $X = \langle x_1, x_2, \ldots \rangle$, `iff` `// indexing assumed to start at 1`
  $\exists i : 1 \leq i \leq |X| \wedge \boldsymbol{x_i} = \boldsymbol{v}$
`element`(X, i, v), `with` $X = \langle x_1, x_2, \ldots \rangle$, `iff`
  $\boldsymbol{x_i} = \boldsymbol{v}$

- The first form of constraint `element` allows us to test the membership of an element in a list.
- The second form allows us to make a connection between a list of variables (or integers) and a variable; this is the usual case.

Remark.
In PyCSP^3, we use natural indexing on lists (see Problem Warehouse).

Semantics

element(X, v), with $X = \langle x_1, x_2, \ldots \rangle$, iff // indexing assumed to start at 1
$\exists i : 1 \leq i \leq |X| \wedge \boldsymbol{x_i} = \boldsymbol{v}$
element(X, i, v), with $X = \langle x_1, x_2, \ldots \rangle$, iff
$\boldsymbol{x_i} = \boldsymbol{v}$

- The first form of constraint `element` allows us to test the membership of an element in a list.
- The second form allows us to make a connection between a list of variables (or integers) and a variable; this is the usual case.

Remark.

In PyCSP^3, we use natural indexing on lists (see Problem Warehouse).

Warehouse Location Problem

Problem 034, proposed by B. Hnich, on CSPLib.

"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."

Data

```
{
  "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]]

}
```

Note that:

- *warehouseCapacities*[*i*] indicates the maximum number of stores that can be supplied by the *ith* warehouse
- *storeSupplyCosts*[*i*][*j*] indicates the cost of supplying the *ith* store with the *jth* warehouse

File Warehouse.py

```
from pycsp3 import *

cost, capacities, costs = data
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) * cost
)
```

Outline

Three possible forms for this constraint:

Semantics

`channel`(X), `with` $X = \langle x_1, x_2, \ldots \rangle$, `iff` `// indexing assumed to start at 1`
$\forall i : 1 \leq i \leq |X|, \boldsymbol{x}_i = j \Rightarrow \boldsymbol{x}_j = i$

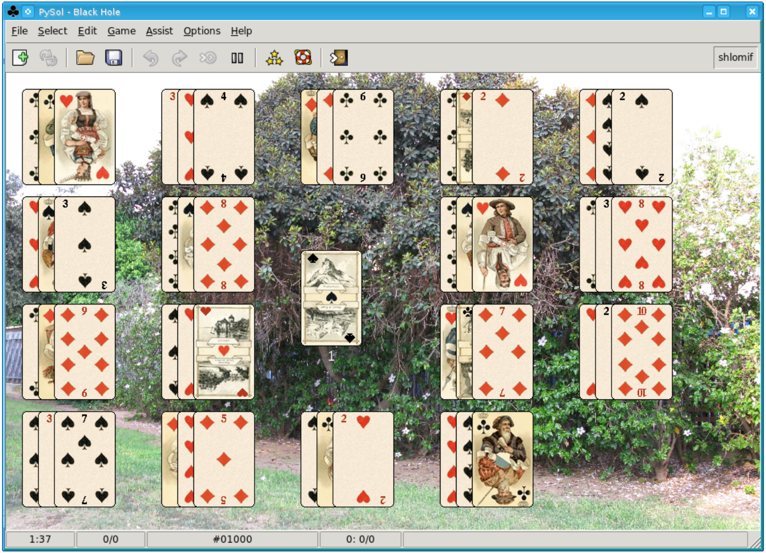
Semantics

`channel`(X, Y), `with` $X = \langle x_1, x_2, \ldots \rangle$ `and` $Y = \langle y_1, y_2, \ldots \rangle$, `iff`
$\forall i : 1 \leq i \leq |X|, \boldsymbol{x}_i = j \Leftrightarrow \boldsymbol{y}_j = i$

Semantics

`channel`(X, v), `with` $X = \{x_1, x_2, \ldots\}$, `iff` `// indexing assumed to start at 1`
$\forall i : 1 \leq i \leq |X|, \boldsymbol{x}_i = 1 \Leftrightarrow \boldsymbol{v} = i$
$\exists i : 1 \leq i \leq |X| \wedge \boldsymbol{x}_i = 1$

Black Hole (solitaire)



Data

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

Note that:

- *piles*[*i*][*j*] indicates the value of the *jth* card on the *ith* pile

File Blackhole.py

```
from pycsp3 import *

m, piles = data  # m denotes the number of cards per suit
nCards = 4 * m

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}


# x[i] is the value j of the card at the ith position 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(
  # 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 must be at a higher or lower rank
  [(x[i], x[i + 1]) in table for i in range(nCards - 1)]
)
```