Logic and Constraint Programming

1- CP Introduction

A.A. 2021/2022

Lorenzo Rossi

lorenzo.rossi@unicam.it

University of Camerino

Why LCP in IAS?

Why LCP in IAS?

Intelligent and Adaptive Systems (IAS) needs to:

- **take decisions according to their knowledge**

So, to program IAS, we need to:

- represent **system's knowledge**
  - **facts** and their **relationships** (i.e., rules, constraints)
- query the knowledge base to support *autonomic decisions*
  - **inference** of an answer to a query, or **solution** of a CSP

Nowadays, other AI supports are available, e.g. Machine Learning

- LCP is programmable and verifiable

An (gentle) Introduction to Constraint Programming

What is CP

Constraint programming (CP) is *paradigm* for solving **combinatorial search problems** that draws on a wide range of techniques from AI, operations research, algorithms, graph theory ...

Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: ***the user states the problem, the computer solves it***[1]

[1] Eugene C. Freuder, Inaugural issue of the Constraints Journal, 1997.

Constraint Satisfaction Problems
≫An example

This is Bob, a Computer Science student at first year

Apart from study, Bob likes eat, play, chill, chat, do sport, travel, and so on

Considering the costs of these activities, and that Bob's parents gives to him 200€/month for all its expenses

What is the maximum number of activities Bob can do with this amount of money?

Constraint Reasoning

Combination

Simplification

Contraddiction

Redundancy

CONSTRAINT SATISFACTION PROBLEMS
≫LET'S TRY TO BE MORE FORMAL

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

Each **row, columns, 3x3 square** must contain numbers **from 1 to 9**, **without repetitions**

Constraint Satisfaction Problems
≫A formal treatment

A constraint satisfaction problem (CSP) is a tuple $\mathcal{X}, \mathcal{D}, \mathcal{C}$ where:

- $\mathcal{X}$ is a set of *variables* $\{X_1, X_2, \ldots, X_n\}$;
- $\mathcal{D}$ is a set of *domains* $\{D_1, D_2, \ldots, D_n\}$ one for each variable; and
- $\mathcal{C}$ is a set of *constraints* over variables.

A domain $D_i = \{v_1, \ldots, v_k\}$ is the set of values allowed for a variable X_1

A constraint C_i is a *relation* over $X_j, \ldots, X_k$

A solution to a CSP is an assignment of values to the variables which satisfies all the constraints simultaneously

CSP Complexity

Given a CSP the *search space* depends on the domains of the variables

$D(X_1) \times \cdots \times D(X_n) \rightarrow$ very large..

Constraint satisfaction is NP-complete

Exist classes of CSP which are tractable, depending on the domains and on the constraints (see [2] for details)

Rossi F., van Beek P., Walsh T.: *Constraint Programming*. In: Handbook of Constraint Programming. Elsevier, 2008.

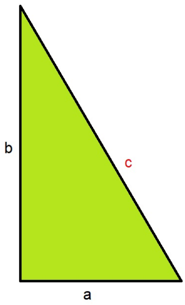
Constraint Satisfaction Problems
≫Pythagorean theorem

Can we consider the Pythagorean theorem as a CSP?

Constraint Satisfaction Problems
≫Pythagorean theorem

Can we consider the Pythagorean theorem as a CSP?

- Variables: $X = \{a, b, c\}$
- Domains: $D(c) = \mathbb{R}^+$
- Constraints: $c^2 = b^2 + a^2$



Minizinc introduction

MINIZINC

MiniZinc is a language designed for specifying constrained optimization and decision problems over **integers** and **real numbers**

A MiniZinc model **does not dictate how to solve the problem** although the model can contain annotations which are used to guide the underlying solver

MiniZinc is designed to interface easily to **different backend solvers**

- An input MiniZinc model and data file is transformed into a FlatZinc model
- FlatZinc models consist of **variable declaration and constraint** definitions as well as a definition of the objective function if the problem is an optimization problem
- The translation from MiniZinc to FlatZinc is specializable to individual backend solvers

MINIZINC

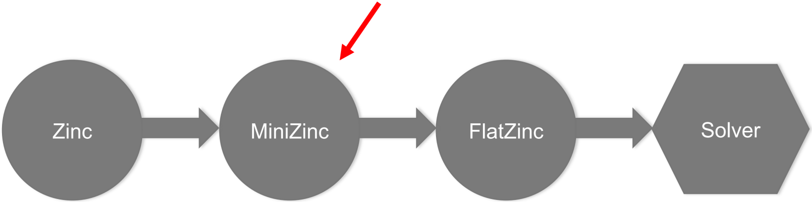
≫SOME MORE INFO

MiniZinc is a high-level, typed, mostly first-order, functional, modelling language. It provides:

- mathematical notation-like syntax (automatic coercions, overloading, iteration, sets, arrays);
- expressive constraints (finite domain, set, linear arithmetic, integer);
- support for different kinds of problems (satisfaction, explicit optimisation);
- separation of data from model;
- extensibility (user-defined functions and predicates);
- reliability (type checking, instantiation checking, assertions);
- **solver-independent modelling**;
- simple, declarative semantics.

From Zinc to FlatZinc

FlatZinc is a low-level solver input language that is the target language for MiniZinc. It is designed to be easy to translate into the form required by a solver



Thus, you can integrate new solvers or implement your own!

MINIZINC
≫SOLVERS

FlatZinc Implementations

- **Gecode/FlatZinc.** The Gecode generic constraint development environment provides a FlatZinc interface. The source code for the interface stripped of all Gecode-specific code is also available.
- **ECLiPSe.** The ECLiPSe Constraint Programming System provides support for evaluating FlatZinc using ECLiPSe's constraint solvers. MiniZinc models can be embedded into ECLiPSe code in order to add user-defined search and I/O facilities to the models.
- **SICStus Prolog.** SICStus (from version 4.0.5) includes a library for evaluating FlatZinc.
- **JaCoP.** The JaCoP constraint solver (from version 4.2) has an interface to FlatZinc.
- **SCIP.** SCIP, a framework for Constraint Integer Programming, has an interface to FlatZinc.
- **Opturion CPX.** Opturion CPX, a Constraint Programming solver with eXplanation system, has an interface to FlatZinc.
- **MinisatID.** MinisatID, an implementation of a search algorithm combining techniques from the fields of SAT, SAT Module Theories, Constraint Programming and Answer Set Programming, has an interface to FlatZinc.

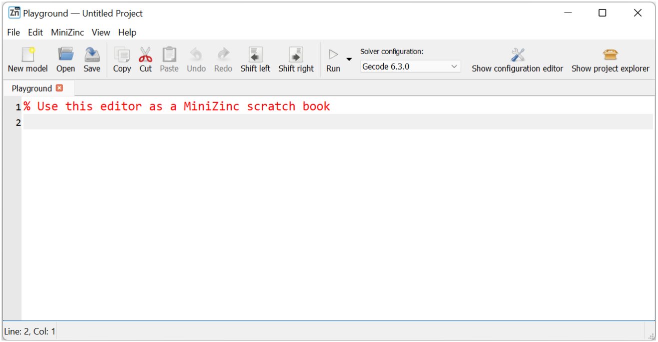
USEFUL RESOURCES

https://www.minizinc.org/

https://www.minizinc.org/doc-2.4.3/en/index.html

A FIRST DEMO

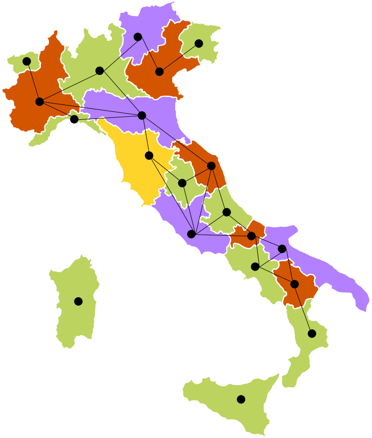
Download and install Minizinc (`https://www.minizinc.org/`)



Example
≫Graph colouring problem

We wish to colour regions in a map. Each region must be coloured so that adjacent regions have different colours

In **graph theory,** graph coloring is a special case of graph labeling; it is an assignment of labels traditionally called "colours" to elements of a graph subject to certain constraints

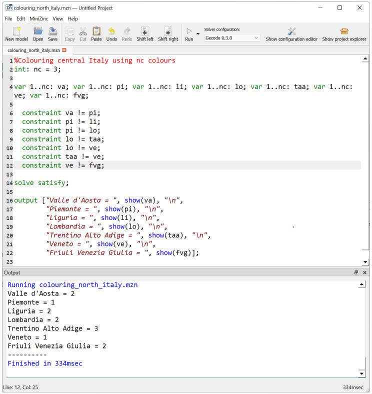


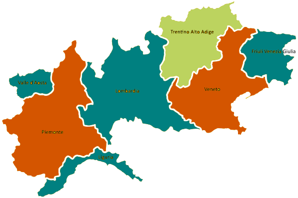
Example
≫Graph colouring problem

Northern Italy includes: Valle d'Aosta, Piemonte, Liguria, Lombardia, Trentino, Friuli, and Veneto region

```
int: nc = 4;
var 1..nc: va; var 1..nc: pi; var 1..nc: li; var 1..nc: lo; var 1..nc: taa; var 1..nc: ve; var 1..nc: fvg;

constraint va != pi;
constraint pi != li;
constraint pi != lo;
constraint lo != taa;
constraint lo != ve;
constraint taa != ve;
constraint ve != fvg;

solve satisfy;

output ["Valle d'Aosta = ", show(va), "\n", "Piemonte = ", show(pi), "\n",
        "Liguria = ", show(li), "\n", "Lombardia = ", show(lo), "\n",
        "Trentino Alto Adige = ", show(taa), "\n", "Veneto = ", show(ve), "\n",
        "Friuli Venezia Giulia = ", show(fvg)];
```

EXAMPLE
≫GRAPH COLOURING PROBLEM





MINIZINC
≫SYNTAX

Comments:

- `%` Single line comment
- `/* */` Comment on multiple lines

Variables: They must have a *type* and be declared. The basic parameter types are integers (**int**), floating point numbers (**float**), booleans (**bool**) and strings (**string**).

- `int: pippo = 3;` Unique declaration and assignment
- `int: pippo; pippo = 3` separated declaration and assignment

Arrays and **sets** are also supported

MINIZINC
≫DECISION AND PARAMETERS

MiniZinc distinguishes between the two kinds of model variables: **parameters** and **decision** variables

- Expressions that can be constructed using decision variables are more restricted than those that can be built from parameters
- In any place that a decision variable can be used, so can a parameter of the same type

The distinction between **parameters** and **decision** variables concerns the instantiation of the variable

- The second is instantiated by **the solver**
- The former is instantiated by **you** (the modeller)

Minizinc
≫Back to the Example

In model we associate a (unknown) decision variable to each region:

var 1..nc: va; **var** 1..nc: pi; **var** 1..nc: li; **var** 1..nc: lo; **var** 1..nc: taa; **var** 1..nc: ve; **var** 1..nc: fvg;

For each decision variable we decide set of possible values the variable can take: **the variable's domain**

In the example we use **integers** to model the different colours.

`1..nc` which is an integer range expression indicating the set `{1, 2, 3}`

Minizinc
≫Back to the Example

The next component of the model are the **constraints** i.e., **boolean expressions that the decision variables must satisfy**

We used *not equal* constraints between the decision variables: **if two states are adjacent then they must have different colours**

```
constraint va != pi;
constraint pi != li;
constraint pi != lo;
constraint lo != taa;
constraint lo != ve;
constraint taa != ve;
constraint ve != fvg;
```

MiniZinc provides:

equal = or ==, not equal != , strictly less than < strictly greater than >, less than or equal to <=, and greater than or equal to >=

Back to the Example

Then, we decide the kind of problem to **solve**

```
solve satisfy;
```

In this case it is a **satisfaction problem**: we wish to find a value for the decision variables that satisfies the constraints but we do not care which one

Finally, we give the **output** statement followed by a list of strings

```
output ["Valle d'Aosta = ", show(va), "\n", "Piemonte = ", show(pi), "\n", "Liguria = ", show(li), "\n
    ", "Lombardia = ", show(lo), "\n", "Trentino Alto Adige = ", show(taa), "\n", "Veneto = ",
    show(ve), "\n", "Friuli Venezia Giulia = ", show(fvg)];
```

String are written between double quotes in a C like notation

Exercise
≫Graph colouring problem

What about central Italy?

Minizinc Syntax

Minizinc
≫Decision variables, Parameters, Types

Minizinc defines parameters and decision variables

```
int:  i=3;
par int:  i=3;
int:  i; i=3;
```

```
var 0..4:  i;
var 0,1,2,3,4:  i;
var int:  i; constraint i >= 0; constraint i <= 4;
```

- Integer: `int` or range `1..n` or `set of int`
- Floating point: `float` or range `1.0 .. n.0` or `set of float`
- Boolean: `bool`
- String: `string` (not for decision variables)
- Array: `array[range] of type`
- Set: `set of type`

SYNTAX
≫STRING

Strings can be only parameters. They are used only for the **output** statement

They are written between double quotes or are expression of the form `show(X)` where `X` can be either a decision variable or a parameter

SYNTAX
≫ARITHMETIC EXPRESSIONS

Operators:

- Float: `* / + -`
- Integer: `* div mod + - abs pow`

Relations:

- `== != > < >= <=`

Minizinc does not provide automatic casting from integer to float. Function `int2float(intexp)` solve this issue

MINIZINC
≫DATA FILES

Model input data can be loaded from file (*.dzn*) or from bash

Model:

```
var int:  A;
int:  B;
```

Data file:

```
A = 12;
B = 2;
```

```
minizinc model.mzn data.dzn
```

Syntax
≫Set

Sets in Minizinc can contain integer, float or Boolean values

---

%Set **of** integer values
set of int: s = {1,23,22,3};

%Set **of** variables
var int: a = 0; **var int**: b = 3;
set of int: s = {a,...,b};

%Range as a **set**
set of int: s = 1..100;

Operators: `in, union, intersect, subset, superset, diff`

A set can be used as a **type**

Syntax
≫Array

Arrays in Minizinc can be multi-dimensional

```
array[index_set1,index_set2, ..., ] of type
```

Index sets of an array can be either:

- Range of integers
- Variable names (representing sets of inegers)

Elements of an array can be of any type excluding other arrays

```
array[products, resources] of int:  consumption;
array[products] of var 0..mproducts:  produce;
```

Exercise
≫Bakery

Bob just opened a bakery in Camerino. Bob knows how to produce two different cakes:

A **banana cake** which takes:

- 250g of flour,
- 2 mashed bananas,
- 75g sugar, and
- 100g of butter.

A **chocolate cake** which takes:

- 200g of flour,
- 75 of cocoa,
- 150g sugar, and
- 150g of butter.

We can sell a chocolate cake for € 4.50 and a banana cake for € 4.00. And we have 4kg of flour, 6 bananas, 2kg of sugar, 500g of butter and 500g of cocoa.

How many of each sort of cake should Bob cook to maximise the profit?

Exercise
≫Bakery

```
% Bob's bakery
var 0..100: b;
var 0..100: c;
%flour
constraint 250*b + 200*c <= 4000;
%bananas
constraint 2*b <= 6;
%sugar
constraint 75*b + 150*c <= 2000;
%butter
constraint 100*b + 150*c <= 500;
%cocoa
constraint 75*c <= 500;
%maximize profit
solve maximize 400*b + 450*c;
output ["Prepare ", show(b), " banana cakes, and ", show(c), "chocolate cakes, now!"];
```