Advanced School in Artificial Intelligence

Bertinoro (FC), 17-28 luglio 2023

Programming with Constraints

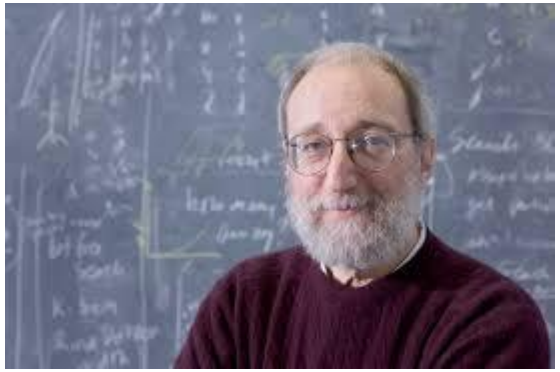
Roberto Amadini

Constraint Programming

- **Constraint Programming (CP)** is a *declarative* paradigm to model and solve CSPs and COPs
- **Declarative** = focus on **what** to solve, rather than how to do it

“ 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 ”

Eugene C. Freuder (1997)
Professor Emeritus, University College Cork



Modelling CP problems

- Converting a real-life problem into a mathematical model that *“better abstracts”* can be **tricky**
  - It requires **expertize**, the concept of “best abstraction” is informal and not univocal

- The same problem can have different yet **equivalent models**
  - the same solver can have **different performance** on equivalent models

- Different solvers can perform differently on the **same model**
  - The user may define a model according to the solver that will solve it
  - *Portfolio solvers*

Encoding CP problems

- Given a mathematical model for a problem, we need to **encode** it in a language understandable by the solver(s) that will solve it
- Officially, **no standard** language to encode CP problems
- However, one of the most known is called **MiniZinc**
  - https://www.minizinc.org/
- MiniZinc is high-level and **solver-independent**
  - *“Model once, solve anywhere”*
- Who developes/developed MiniZinc?
  - **Monash University**, CSIRO Data61, University of Melbourne

MiniZinc

- MiniZinc is modelling language, **not a solver!** It allows the user to specify:
  - **Parameters**
    - MiniZinc also provides **separation** model/data
  - **Variables** of different **type**, and corresponding **domains**
    - Boolean, integers, floats, set of integers, …
  - **Constraints** over the variables
    - Arithmetical, logical, **global**
  - **Objective** (minimization/maximization)
  - …and much more!

Example: Sudoku

```
include "globals.mzn";

function array[int] of var int:
subgrid(array[int,int] of var int: grid, int: i, int: j) =
  [grid[3*(i-1)+p, 3*(j-1)+q] | p in 1..3, q in 1..3];

array[1..9,1..9] of var 1..9: grid = [|
  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
|];
constraint forall (i in 1..9) (all_different([grid[i,j] | j in 1..9]));
constraint forall (j in 1..9) (all_different([grid[i,j] | i in 1..9]));
constraint forall (i in 1..3, j in 1..3) (all_different(subgrid(grid, i,j)));

solve satisfy;
```

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

Example: Sudoku

Output

```
Finished in 127msec
Compiling sudoku.mzn
Running sudoku.mzn
```

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

```
----------
Finished in 130msec
```

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

Example: Subset-sum

- **subset-sum problem**: are there **N** numbers in a set **S** adding up to **K**?

```
% Are there N numbers in a set S adding up to K?
include "globals.mzn";

set of int: S = {7, 10, 23, 13, 4, 16};
int: N = 4;
int: K = 50;

array[1..N] of var S: X;
constraint all_different(X);
constraint sum(X) = K;

solve satisfy;
```

Example: Subset-sum

- **subset-sum problem**: are there **N** numbers in a set **S** adding up to **K**?

```
% Are there N numbers in a set S adding up to K?
include "globals.mzn";

set of int: S = {7, 10, 23, 13, 4, 16};
int: N = 4;
int: K = 50;

array[1..N] of var S: X;
constraint all_different(X);
constraint sum(X) = K;

solve satisfy;
```

For using **global constraints**

Example: Subset-sum

- **subset-sum problem**: are there **N** numbers in a set **S** adding up to **K**?

```
% Are there N numbers in a set S adding up to K?
include "globals.mzn";

set of int: S = {7, 10, 23, 13, 4, 16};
int: N = 4;
int: K = 50;

array[1..N] of var S: X;
constraint all_different(X);
constraint sum(X) = K;

solve satisfy;
```

For using **global constraints**

Parameters

Example: Subset-sum

- **subset-sum problem**: are there **N** numbers in a set **S** adding up to **K**?

```
% Are there N numbers in a set S adding up to K?
include "globals.mzn";

set of int: S = {7, 10, 23, 13, 4, 16};
int: N = 4;
int: K = 50;

array[1..N] of var S: X;
constraint all_different(X);
constraint sum(X) = K;

solve satisfy;
```

For using **global constraints**

Parameters

Instead of **N** integer variables **X_1, …, X_N** we use an **array X** of **N** integer variables

Example: Subset-sum

- **subset-sum problem**: are there **N** numbers in a set **S** adding up to **K**?

```
% Are there N numbers in a set S adding up to K?
include "globals.mzn";

set of int: S = {7, 10, 23, 13, 4, 16};
int: N = 4;
int: K = 50;

array[1..N] of var S: X;
constraint all_different(X);
constraint sum(X) = K;

solve satisfy;
```

For using **global constraints**

Parameters

Instead of **N** integer variables **X_1, …, X_N** we use an **array X** of **N** integer variables

Global constraints: defined on arbitrary number of variables

Example: Subset-sum

- **subset-sum problem**: are there **N** numbers in a set **S** adding up to **K**?

```
% Are there N numbers in a set S adding up to K?
include "globals.mzn";

set of int: S = {7, 10, 23, 13, 4, 16};
int: N = 4;
int: K = 50;

array[1..N] of var S: X;
constraint all_different(X);
constraint sum(X) = K;

solve satisfy;
```

For using **global constraints**

Parameters

Instead of **N** integer variables $\mathbf{X_1, \ldots, X_N}$ we use an **array X** of **N** integer variables

Global constraints: defined on arbitrary number of variables

CSP

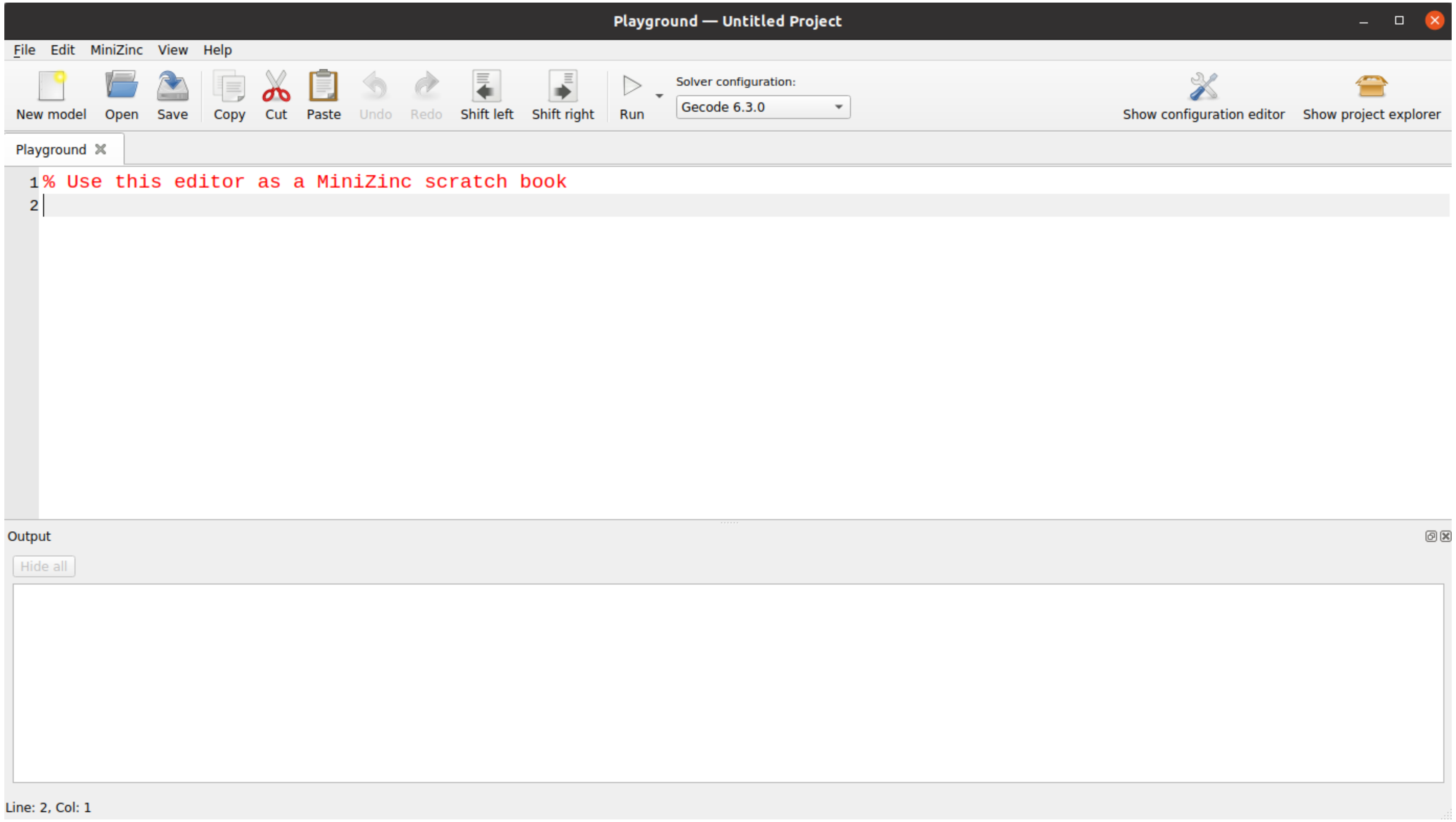
Getting started

- **Download** and Install **MiniZinc**: https://www.minizinc.org/software.html
  - Bundled binary packages recommended

- **MiniZinc IDE**: Integrated Development Environment to:
  - **Develop** MiniZinc models (editor)
  - **Compile** MiniZinc models into FlatZinc, a low-level language understood by a large range of solvers
    - Solvers solve the derived FlatZinc, not the MiniZinc model
  - **Solve** a compiled model by one of the integrated solvers
    - Chuffed
    - Gecode
    - Coin-BC
    - …

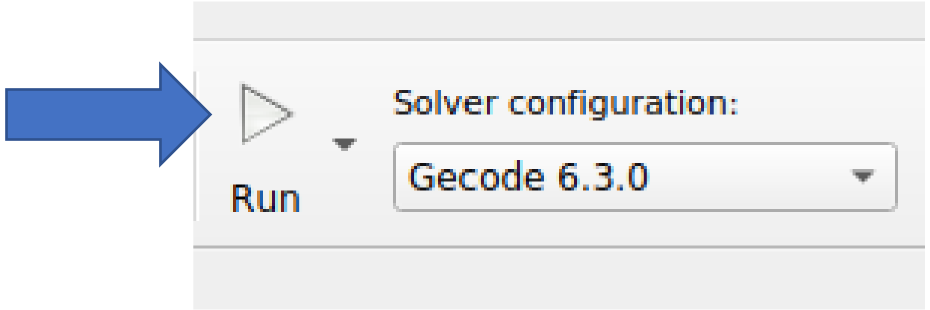
Getting Started

- Now open the **MiniZinc IDE**. It should appear something like:

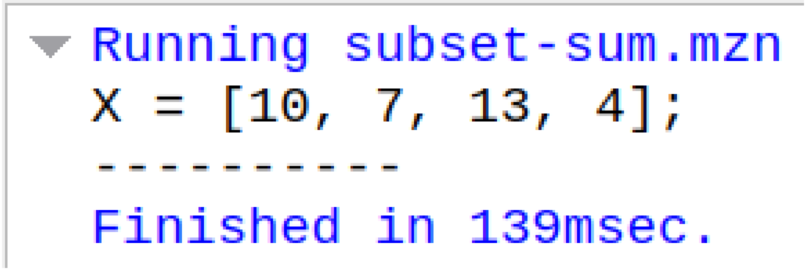


MiniZinc IDE

- **Exercise**: Implement the above models (or other models!) with MiniZinc and solve them
  - The FlatZinc compilation is transparent for the user, just pick a solver and click "**Run**"!



- Use **different solvers**:
  - Chuffed
  - COIN-BC
  - Gecode
  - …
- …Is the **output** always the same?



Solving CP problems

- Once a CP model is defined, a **constraint solver** is used to solve the constraints and (possibly) return a solution

- CP solving basically works in two steps:

  - **Propagation**: the domains of the variables are **pruned** until no more pruning is possible (not complete)
    - E.g., propagating $x < y$ with $D(x) = [1,5]$, $D(y) = [-2,4]$ results in $D(x) \leftarrow [1,3]$ and $D(y) \leftarrow [2,4]$. This in turn may trigger other propagators until a **fixpoint** is reached

  - **Search**: we "guess" the value of a variable (heuristics) and if we have a failure we **backtrack**, until either all the variables are **assigned** (we have a solution) or **unsatisfiability** is proven (all the alternatives fail)

Solving CP problems

- A powerful technique for solving (not only) CP problems is called **clause learning**
  - a.k.a. **no-goods** learning
  - Basically, redundant constraints are **learned** during the solving process to **avoid to repeat** the same "*bad choices*" during the search process

- Examples of effective CP solvers using clause learning are **Chuffed** (part of MiniZinc bundle), **OR-Tools** (developed by Google), and **Opturion** (commercial software)
  - https://github.com/chuffed/chuffed
  - https://developers.google.com/optimization
  - https://www.opturion.com/

- Other well-known CP solvers: Gecode, iZplus, Picat, Choco, etc…
  - See *MiniZinc Challenge*: https://www.minizinc.org/challenge.html

Exercise

- We are **master brewers**. We bought the right ingredients (Corn, Hop, Malt) and we need to **decide** how many **Ales** and **Beers** as possible, given the **resources available,** to maximize the potential **profit**:

Beverage	Corn	Hops	Malt	Profit
Ale	5	4	35	13
Beer	15	4	20	23
Q.ty available	480	160	1190	

- First define a **model** for this problem
  - Identify variables (decisions), domains (options), constraints (requirement), objective function (goal)
- Then **implement** it and **solve** it with MiniZinc
  - *Hint*: use **solve maximize** instead of solve satisfy…