



Introduction to MinZinc





CP - Declarative Programming

Procedural:

- 1 Go to kitchen
- 2 Get Tea leaf and water
- 3 Mix them and heat over fire till it boils
- 4 Put that in a cup and bring it to me

e.g. Python, C, Java

Declarative:

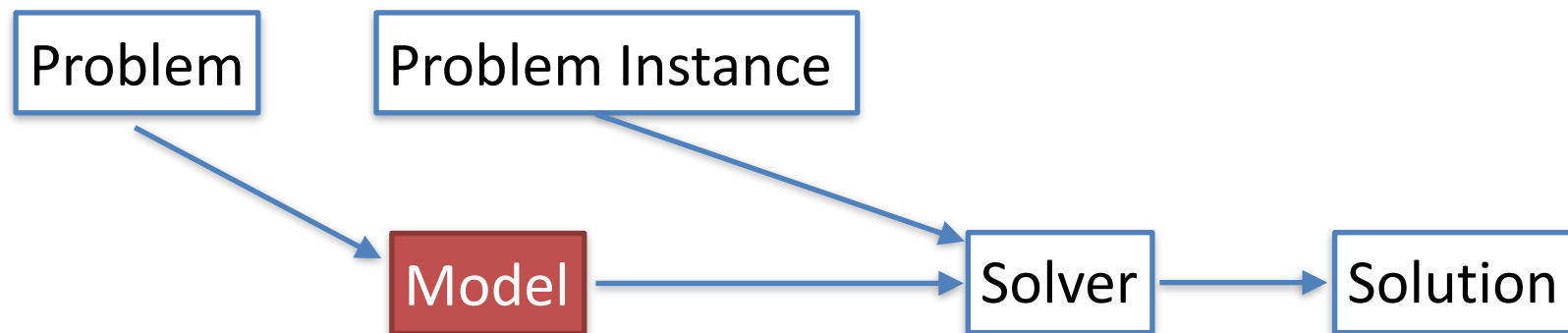
- 1 Tea is composed by tea leaf with hot water
- 2 Get me a cup of tea.

Minizinc

Declarative programming is where you describe what you want without having to say how to do it. With procedural programming, you have to specify exact steps to get the result.

CP - Methodology


Model Problem - Solve Model



state (describe) the problem



CP - How to describe a problem to Computer?

- Entities (Variables) in the problem and who are they?
 - number, string, set of (number/string)
 - the domain of Entities
 - Relation (Constraints) between the Entities and what entity should look like
 - My Goal (Objective)
 - e.g. planing a tour
 - Entities: city, distance between cities ...
 - Relation: I won't visit a city twice, I start from city x ...
 - Goal: I want to save my time
 - => a path that satisfies the constraints
 - => Formally, we describe it as a CSP (and COP)
- 




CSP - Constraint Satisfaction Problem

- A CSP is defined by
 - a finite set of variables $\{X_1, \dots, X_n\}$
 - a set of values(domain) for each variable $dom(X_1), \dots, dom(X_n)$
 - A set of constraints $\{C_1, \dots, C_m\}$
- A **solution** to a CSP is a complete assignment to all the variables that satisfies the constraints.

COP - Constraint Optimization Problem

A COP is a CSP defined on the variables x_1, \dots, x_n , together with an objective function $f : D(x_1) \times \dots \times D(x_n) \rightarrow Q$ that assigns a value to each variable. An optimal solution to a COP is a solution d that optimize the value of $f(d)$.

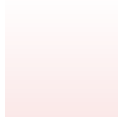


Model CSP with Minizinc

Minizinc, a well-known Constraint Modeling Language
which is becoming the standard



- Mainly at U of Melbourne and Monash U, Australia
- Introduced in 2007, v 2.1.7 in 2018
- Homepage: <http://www.minizinc.org>.
- Courses available also in [Coursera](#)



Some features of Minizinc

- Each expression terminates with ‘;’
- Variable domain, array index domain must all be specified
 - a domain could be 1..10 (Yes, ‘..’, not ‘...’) or int or an array
- Keyword ‘constraint’ denotes the rules that a solution should meet
- Index starts from ‘1’ not ‘0’
- ‘%’ for comments

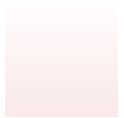
```
1 include "globals.mzn";
2
3 int: n;
4 array[1..n,1..n] of int: dist;
5 int: start_city;
6 int: end_city;
7 array[1..n] of var 1..n: city;
8 array[1..n] of string: city_name;
9
10 constraint city[1] = start_city;
11 constraint city[n] = end_city;
12 constraint all_different(city);
```

Import global constraint library
in order to use ‘all_different()’

Parameter (not variable),
its value will be given by problem instance

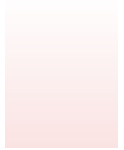
Variable (a.k.a decision variable),
its domain values will be checked
in order to find solution

Constraints specify your requirements
e.g. I start and end up my trip in specific cities



Minizinc Basics: Data Representation

- Parameters - values are passed by problem instance
- `[domain]` : [parameter name]
- e.g. `int: n = 10;`
- Variables - values depend on the solution
- `var [domain]` : [variable name]
- e.g. `var int: total_distance; % traveling distance in a tour`
- Arrays - can be a set of either parameters or variables
- `array[index_domain]` of `[domain]` %parameter array
- `array[index_domain]` of `var [domain]` %variable array
- e.g.
- `array[1..n]` of `var 1..n`: city; % the order of city I visit from 1 to n
- `array[1..n,1..n]` of `int`: distance; % distance between a pair of city



Minizinc Basics: Aggregation Functions

- **sum, product, min, max**
- e.g.
 - `sum(array_x)`, % sum of all the elements in array_x
 - `sum(i in 1..3)(array_x[i])`, % sum of elements from 1 to 3 of array_x
- **forall** (*counter(s) in domain*) (*constraints*),
- **forall** (*counter(s) in domain where condition*) (*constraints*)
- e.g.
 - **forall** (i,j **in** 1..3 **where** i < j) (array_x[i] != array_x[j]) % first 3 elements in array_x are different
- **exists** (*counter(s) in domain*) (*constraints*)
- and others ...



Minizinc Basics: Constraints

- Constraints are rules that a solution must respect
- **constraints** [expression]
- e.g.
- **constraint** city[1] = start_city
- **constraint** all_different(city);
- all_different is a global constraint, which deals with an arbitrary number of variables. Global constraints are notations easily recognized by CP solvers where efficient solving techniques will be applied.
- global constraints are also in fact composed by basic constraints:

```
all_different(array x) =  
forall ( i, j in index(x) where i < j ) ( x[i] != x[j] )
```



Minizinc Basics: Constraints Language

- Arithmetic Operators
 - $+$, $-$, $*$, $/$, $^$, $=$, \neq
 - e.g. $x + y^2 = z$
- Logical Operators
 - \vee , or
 - \wedge , and,
 - constraint $A \wedge B$; means also constraint A; constraint B;
 - \rightarrow , implies
 - $!$ negation
 - e.g. $x = 1 \rightarrow y \neq 5$ % if x equals to 1 then y must not be 5
- Global Constraints
 - `all_different()`
 - `all_equal()` ...

Two type of solutions:

- 1) Satisfiability problems: we simply call the solver to find one solution. We write in the code

`solve satisfy`

- 2) Optimization problems: we want to computer the best solution which optimize a given function: see next slide

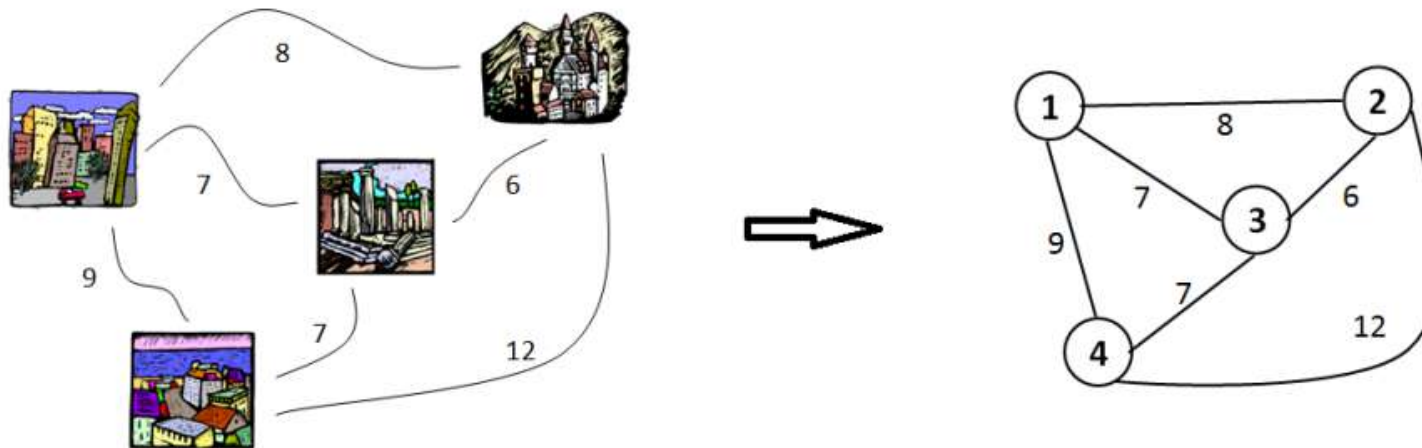


Minizinc Basics: Objective



- define the objective
- define the aim of solution
- `var int: total_distance = sum(i in 2..n) (dist [city[i-1], city[i]]);`
- `solve minimize total_distance;`

Understanding Minizinc with problems



Traveling Salesman Problem:
I visit each city once and I want to save my time

Complete Code - Traveling Salesman Problem



```
1 include "globals.mzn";
2
3 int: n;
4 array[1..n,1..n] of int: dist;
5 int: start_city;
6 int: end_city;
7
8 array[1..n] of var 1..n: city;
9 array[1..n] of string: city_name;
10
11 constraint city[1] = start_city;
12 constraint city[n] = end_city;
13 constraint all_different(city);
14
15 var int: total_distance = sum(i in 2..n)(dist[city[i-1],city[i]]);
16 solve minimize total_distance;
17
18 output [city_name[fix(city[i])] ++ " -> " | i in 1..n ] ++
19         ["\nTotal hours travelled: ", show(total_distance) ];
20
```

Just state
your problem

Problems solved with Minizinc



- factory scheduling (JSSP)
- vehicle routing (VRP)
- packing problems (NumPart and BinPack)
- timetabling (exams, lectures, trains)
- configuration and design (hardware)
- workforce management (call centres, etc)
- car sequencing (assembly line scheduling)
- supertree construction (bioinformatics)
- network design (telecoms problem)
- gate arrival (at airports)
- logistics (Desert Storm an example)
- aircraft maintenance schedules
- aircraft crew scheduling (commercial airlines)
- air cover for naval fleet

An investment problem

- 225K available
- 28 persons av.
- max 9 projects

Project name	Value	Budget (k)	Personnel
Ischia	6000	35	5
Speltra	4000	34	3
Hotel	1000	26	4
Restaurant	1500	12	2
ContoA	800	10	2
ContoB	1200	18	2
Scuola	2000	32	4
Dago	2200	11	1
Lago	900	10	1
small	3800	22	5
Iper	2900	27	3
Bivio	1300	28	2
Tela	800	16	2
Idro	2700	29	4
Batment	2800	22	3



An investment problem

- *Some projects cannot be selected with some others*
- *Some projects require some other projects*

Project name	Value	Budget	Personnel	Not With	Require
Ischia	6000	35	5	10	-
Speltra	4000	34	3	-	-
Hotel	1000	26	4	-	15
Restaurant	1500	12	2	-	15
ContoA	800	10	2	6	-
ContoB	1200	18	2	5	-
Scuola	2000	32	4	-	-
Dago	2200	11	1	-	7
Lago	900	10	1	-	-
small	3800	22	5	1	-
Iper	2900	27	3	15	-
Bivio	1300	28	2	-	-
Tela	800	16	2	-	2
Idro	2700	29	4	-	2
Batment	2800	22	3	11	-

```

int: num_projects; % number of projects to select from
int: max_budget; % budget limit
int: max_persons; % persons available
int: max_projects; % max number of projects to select
% the values of each project
array[1..num_projects] of int: values;
array[1..num_projects] of int: budgets;
array[1..num_projects] of int: personnel;
% project i cannot be selected with project j
int: num_not_with;
array[1..num_not_with, 1..2] of 1..num_projects: not_with;
% project i requires project j
int: num_requires;
array[1..num_requires, 1..2] of 1..num_projects: requires;
% decision variable: what project to select
array[1..num_projects] of var 0..1: x;
var int: total_persons = sum(i in 1..num_projects) (x[i]*personnel[i]);
var int: total_budget = sum(i in 1..num_projects) (x[i]*budgets[i]);
var int: total_projects = sum(i in 1..num_projects) (x[i]);
constraint
    total_budget <= max_budget
    /\
    total_persons <= max_persons
    /\
    total_projects <= max_projects
;

var int: total_values = sum(i in 1..num_projects) (x[i]*values[i]);

solve maximize total_values;

```



More resources

Minizinc Tutorial:

<http://www.minizinc.org/downloads/doc-latest/minizinc-tute.pdf>

<https://www.minizinc.org/downloads/doc-1.2/minizinc-tute.pdf>

Coursera:

<https://www.coursera.org/learn/basic-modeling>

More Minizinc Examples:

<https://github.com/hakank/hakank/tree/master/minizinc>

<https://github.com/MiniZinc/minizinc-examples>

Credits: Hakank, Chiarandini, Peter van BeeK

Exercises



A First MinZinc program

```
% Find 4 numbers such that their sum is 711 and their product is  
711*10*10*10,
```

```
var 1..711: item1;
```

```
var 1..711: item2;
```

```
var 1..711: item3;
```

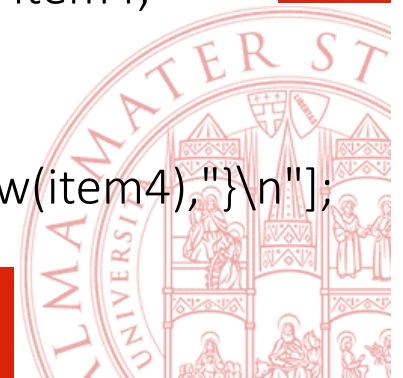
```
var 1..711: item4;
```

```
constraint item1 + item2 + item3 + item4 == 711; constraint item1 * item2 * item3 *  
item4 == 711 * 100 * 100 * 100;
```

```
constraint 0 < item1 /\ item1 <= item2 /\ item2 <= item3 /\ item3 <= item4;
```

```
solve satisfy;
```

```
output ["{", show(item1), ",", show(item2), ",", show(item3), ",", show(item4), "}" \n];
```



Cakes problem

We need to bake some cakes for a fete at our local school.

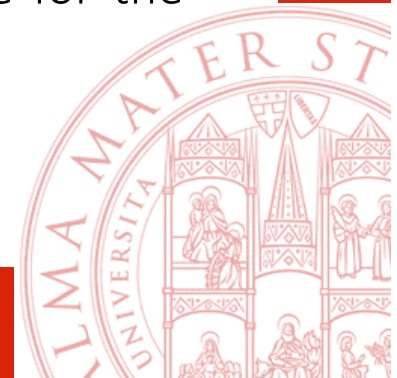
We know how to make two sorts of cakes.

- 1) A banana cake which takes 250g of self-raising flour, 2 mashed bananas, 75g sugar and 100g of butter,
- 2) A chocolate cake which takes 200g of self-raising flour, 75g of cocoa, 150g sugar and 150g of butter.

We can sell a chocolate cake for \$4.50 and a banana cake for \$4.00.

We have 4kg self-raising flour, 6 bananas, 2kg of sugar, 500g of butter and 500g of cocoa.

The question is how many of each sort of cake should we bake for the fete in order to maximize the profit.



Example 1 from Minizinc Tutorial: Cakes

CAKES ≡

[\[DOWNLOAD\]](#)

```
% Baking cakes for the school fete

var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes

% 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 our profit
solve maximize 400*b + 450*c;

output ["no. of banana cakes = \"(b)\\n\",
        "no. of chocolate cakes = \"(c)\\n\"];
```

Figure 3: Model for determining how many banana and chocolate cakes to bake for the school fete.



Example 2 from Minizinc Tutorial

Another powerful modelling feature in MiniZinc is that decision variables can be used for array access. As an example, consider the (old-fashioned) *stable marriage problem*. We have n (straight) women and n (straight) men. Each man has a ranked list of women and vice versa. We want to find a husband/wife for each women/man so that all marriages are *stable* in the sense that:

- whenever m prefers another women o to his wife w , o prefers her husband to m , and
- whenever w prefers another man o to her husband m , o prefers his wife to w .



STABLE-MARRIAGE \equiv

[\[DOWNLOAD\]](#)

```
int: n;
```

```
enum Men = anon_enum(n);
```

```
enum Women = anon_enum(n);
```

```
array[Women, Men] of int: rankWomen;
```

```
array[Men, Women] of int: rankMen;
```

```
array[Men] of var Women: wife;
```

```
array[Women] of var Men: husband;
```

► **ASSIGNMENT**

► **RANKING**

```
solve satisfy;
```

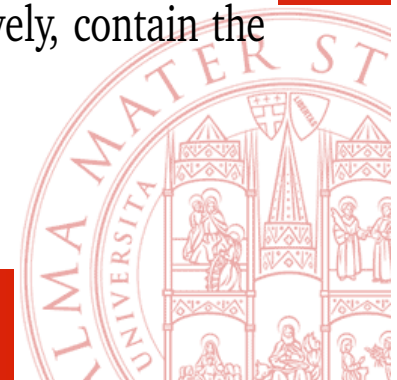
```
output ["wives= \$(wife)\nhusbands= \$(husband)\n"];
```



The first three items in the model declare the number of men/women and the set of men and women. Here we introduce the use of *anonymous enumerated types*. Both Men and Women are sets of size n , but we do not wish to mix them up so we use an anonymous enumerated type. This allows MiniZinc to detect modelling errors where we use Men for Women or vice versa.

The matrices rankWomen and rankMen, respectively, give the women's ranking of the men and the men's ranking of the women. Thus, the entry rankWomen[w,m] gives the ranking by woman w of man m. The lower the number in the ranking, the more the man or women is preferred.

There are two arrays of decision variables: wife and husband. These, respectively, contain the wife of each man and the husband of each women.



ASSIGNMENT \equiv

```
constraint forall (m in Men) (husband[wife[m]]=m);  
constraint forall (w in Women) (wife[husband[w]]=w);
```

ensure that the assignment of husbands and wives is consistent: w is the wife of m implies m is the husband of w and vice versa. Notice how in `husband[wife[m]]` the index expression `wife[m]` is a decision variable, not a parameter.

The next two constraints are a direct encoding of the stability condition:

RANKING \equiv

```
constraint forall (m in Men, o in Women) (  
    rankMen[m,o] < rankMen[m,wife[m]] ->  
    rankWomen[o,husband[o]] < rankWomen[o,m] );  
  
constraint forall (w in Women, o in Men) (  
    rankWomen[w,o] < rankWomen[w,husband[w]] ->  
    rankMen[o,wife[o]] < rankMen[o,w] );
```

