# Knowledge Representation and Reasoning Introduction to Constraint Programming with MiniZinc – 2

Antoni Ligęza

ligeza@agh.edu.pl

AGH University of Science and Technology
Kraków, Poland

Knowledge Representation and Reasoning
AGH Kraków
2017

# Outline

# Presentation Outline

# Why Arrays and Sets?

- the number of variables can change with the size of the problem:
  - e.g. number of products (array: quantity of product),
  - e.g. number of rates to be paid (array: amount of rate),
  - e.g. number of components/blocks (array: component value),
- **array** can be of one, two, and many dimensions,
- notation `temp[3,7]`; a variable associated with the position `(3,7)` on a grid,
- one can define a single constraint over an array – all the variables!
- `array[1..5] = [1,3,5,7,11]` – one-dimensional array,
- `array[1..3,1..4] = [|1,2,3|,|4,5,6|,|7,8,9|,|10,11,12|]` – two-dimensional $4 \times 3$ array,
- `constraint forall(i in 1..w-1)(temp[i,h] = right);` – –example constraint over a border,
- list comprehension: special form of list specification, e.g. `forall( [a[i] != a[j] | i,j in 1..3 where i < j])`

# Laplace: a Visualization

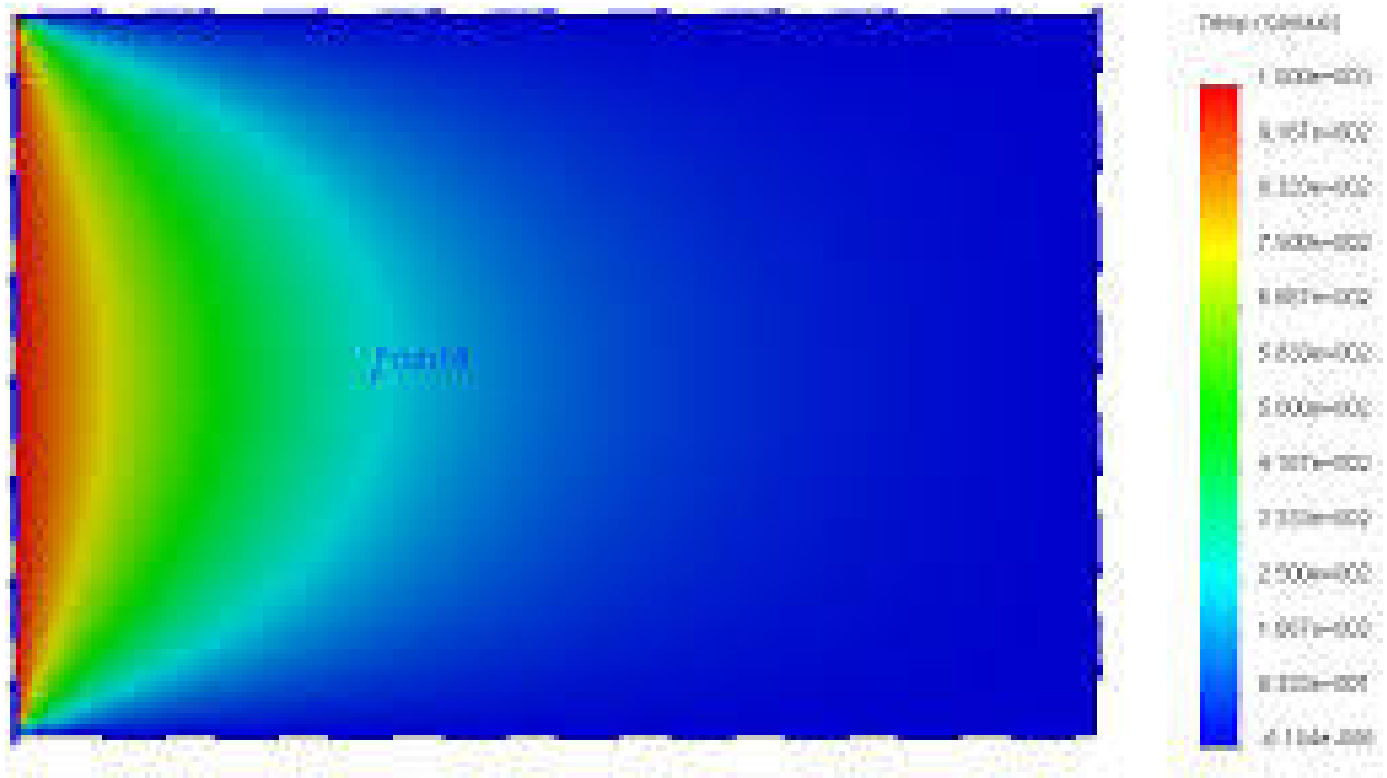

**Figure :** A Visualization of 2-D Temperature Distribution

# MiniZinc Coding: Laplace I

```
int: w = 4;
int: h = 4;

% arraydec
array[0..w,0..h] of var float: t; % temperature at point (i,j)
var float: left;   % left   edge temperature
var float: right;  % right  edge temperature
var float: top;    % top    edge temperature
var float: bottom; % bottom edge temperature

% equation
% Laplace equation: each internal temp.
% is average of its neighbours
constraint forall(i in 1..w-1, j in 1..h-1)(
   4.0*t[i,j] = t[i-1,j] + t[i,j-1] + t[i+1,j] + t[i,j+1]);
% sides
% edge constraints
```

# MiniZinc Coding: Laplace II

```
constraint forall(i in 1..w-1)(t[i,0] = left);
constraint forall(i in 1..w-1)(t[i,h] = right);
constraint forall(j in 1..h-1)(t[0,j] = top);
constraint forall(j in 1..h-1)(t[w,j] = bottom);
% corners
% corner constraints
constraint t[0,0]=0.0;
constraint t[0,h]=0.0;
constraint t[w,0]=0.0;
constraint t[w,h]=0.0;
left = 0.0;
right = 0.0;
top = 100.0;
bottom = 0.0;

solve satisfy;

output [ show_float(6, 2, t[i,j]) ++
```

# MiniZinc Coding: Laplace III

```
        if j == h then "\n" else " " endif |
        i in 0..w, j in 0..h
];
```

# Aggregation Functions

**Arithmetic aggregation**

- `sum()` – summation of elements on a list,
- `product()` – multiplication of elements on a list,
- `min()` – minimal element from a list,
- `max()` – maximal element on a list.

**Logical aggregation (on arrays of Boolean expressions)**

- `forall` – logical conjunction of expressions of an array,
- `exists` – logical disjunction of expressions of an array.

# Using Sets

Sets are (unordered!) collections of items. Some features and use:

- sets can be of the following types: integers, floats, and Booleans,
- sets can be (and typically are (ordered!) *range expressions* of the form: `FIRST..LAST`,
- `set of int: Products = 1..nproducts` – declaration of a set of int,
- sets of literals are allowed $\{e_1, \ldots, e_k\}$,
- standard operations: `in, subset, superset, union, inter, diff, symdiff; card`.

# MiniZinc Coding: Cakes Revisited I

```
% Number of different products
int: nproducts;
set of int: Products = 1..nproducts;
% profit per unit for each product
array[Products] of int: profit;
array[Products] of string: pname;
% Number of resources
int: nresources;
set of int: Resources = 1..nresources;
% amount of each resource available
array[Resources] of int: capacity;
array[Resources] of string: rname;

% units of each resource required to produce 1 unit of product
array[Products, Resources] of int: consumption;
constraint assert(forall (r in Resources, p in Products)
      (consumption[p,r] >= 0), "Error: negative consumption");
```

# MiniZinc Coding: Cakes Revisited II

```
% bound on number of Products
int: mproducts = max (p in Products)
                (min (r in Resources where consumption[p,r] > 0)
                     (capacity[r] div consumption[p,r]));

% Variables: how much should we make of each product
array[Products] of var 0..mproducts: produce;
array[Resources] of var 0..max(capacity): used;

% Production cannot use more than the available Resources:
constraint forall (r in Resources) (
 used[r] = sum (p in Products)(consumption[p, r] * produce[p])
      /\ used[r] <= capacity[r]
);

% Maximize profit
solve maximize sum (p in Products) (profit[p]*produce[p]);
```

# MiniZinc Coding: Cakes Revisited III

```
output [ show(pname[p]) ++ " = " ++ show(produce[p]) ++ ";\n" |
         p in Products ] ++
       [ show(rname[r]) ++ " = " ++ show(used[r]) ++ ";\n" |
         r in Resources ];
```

# Global Constraints I

```
include "alldifferent.mzn";

var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;

constraint           1000 * S + 100 * E + 10 * N + D
                   + 1000 * M + 100 * O + 10 * R + E
       = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint alldifferent([S,E,N,D,M,O,R,Y]);
```

# Global Constraints II

```
solve satisfy;

output ["   ",show(S),show(E),show(N),show(D),"\n",
        "+  ",show(M),show(O),show(R),show(E),"\n",
        "= ",show(M),show(O),show(N),show(E),show(Y),"\n"];
```

# Conditional Expressions

Conditional expressions: *if COND then EXPR1 else EXPR2 endif;*
Example:

```
int:  r = if y != 0 then x div y else 0 endif;
```

Example – Sudoku initialization:

```
constraint forall(i,j in PuzzleRange)
(if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif
);
```

# MiniZinc Coding: Sudoku I

```
include "alldifferent.mzn";

int: S;
int: N = S * S;
int: digs = ceil(log(10.0,int2float(N))); % digits for output

set of int: PuzzleRange = 1..N;
set of int: SubSquareRange = 1..S;

array[1..N,1..N] of 0..N: start; %% initial board 0 = empty
array[1..N,1..N] of var PuzzleRange: puzzle;

% fill initial board
constraint forall(i,j in PuzzleRange)(
   if start[i,j] > 0 then puzzle[i,j] = start[i,j]
   else true endif );
```

# MiniZinc Coding: Sudoku II

```
% All different in rows
constraint forall (i in PuzzleRange) (
   alldifferent( [ puzzle[i,j] | j in PuzzleRange ]) );

% All different in columns.
constraint forall (j in PuzzleRange) (
   alldifferent( [ puzzle[i,j] | i in PuzzleRange ]) );

% All different in sub-squares:
constraint
   forall (a, o in SubSquareRange)(
      alldifferent( [ puzzle[(a-1) *S + a1, (o-1)*S + o1] |
                      a1, o1 in SubSquareRange ] ) );

solve satisfy;

output  [ show_int(digs,puzzle[i,j]) ++ " " ++
          if j mod S == 0 then " " else "" endif ++
```

# MiniZinc Coding: Sudoku III

```
if j == N /\ i != N then
    if i mod S == 0 then "\n\n" else "\n" endif
else "" endif
| i,j in PuzzleRange ] ++ ["\n"];
```

# Complex Constraints I

- `/\` – concjunction,
- `\/` – disjunction,
- `->` – implication,
- `<-` – only-if (implication to the left),
- `<->` – if-and-only-if (equivalence),
- `not` – negation.

Example:

```
constraint s1 + d1 <= s2 \/ s2 + d2 <= s1;
```

s – start of a task,
d – duration of a task
(execution of tasks cannot overlap on a single machine)
The Job-Shop example:

- a set of job must be completed,
- each job consists of sequential tasks,
- the tasks must be executed in order,
- and on separate machines.

`bool2int` – a function convert Boolean to integer (true $= 1$, false $= 0$).

# MiniZinc Coding: A Job-Shop Example I

```
int: jobs;                                          % no of jobs
int: tasks;                                         % no of tasks per job
array [1..jobs,1..tasks] of int: d;                 % task durations
int: total = sum(i in 1..jobs, j in 1..tasks)
             (d[i,j]);                              % total duration
int: digs = ceil(log(10.0,int2float(total)));       % digits for output
array [1..jobs,1..tasks] of var 0..total: s;        % start times
var 0..total: end;                                  % total end time


constraint %% ensure the tasks occur in sequence
    forall(i in 1..jobs) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,tasks] + d[i,tasks] <= end
    );
```

# MiniZinc Coding: A Job-Shop Example II

```
constraint %% ensure no overlap of tasks
    forall(j in 1..tasks) (
        forall(i,k in 1..jobs where i < k) (
            s[i,j] + d[i,j] <= s[k,j] \/
            s[k,j] + d[k,j] <= s[i,j]
        )
    );

solve minimize end;

output ["end = ", show(end), "\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in 1..jobs, j in 1..tasks ];
```

# Example: Stable Marriage

In MiniZinc decision variables can be used for array access.

## Stable Marriage

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.

# MiniZinc Coding: Stabe Marriage I

```
int: n;

set of int: Men = 1..n;
set of int: Women = 1..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
constraint forall (m in Men) (husband[wife[m]]=m);
constraint forall (w in Women) (wife[husband[w]]=w);
% ranking
constraint forall (m in Men, o in Women) (
     rankMen[m,o] < rankMen[m,wife[m]] ->
```

# MiniZinc Coding: Stabe Marriage II

```
        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] );
solve satisfy;

output ["wives= ", show(wife),"\n", "husbands= ", show(husband), "\
```

# Presentation Outline

# Higher-Order Constraints

## Magic Series Problem

Given $n$, find a sequence (a list) of numbers

$$s = [s_0, s_1, \ldots, s_{n-1}]$$

such that:

$$s_i = Number\_of\_occurrences(i)$$

An example is s = [1, 2, 1, 0].

Higher-order constraint: the function `bool2int` takes as its argument an arbitrary boolean expression; the expression is evaluated to true/false and the results is converted to integer.

# MiniZinc Coding: Magic Series Example I

```
int: n;
array[0..n-1] of var 0..n: s;

constraint forall(i in 0..n-1) (
   s[i] = (sum(j in 0..n-1)(bool2int(s[j]=i))));

solve satisfy;

output [ "s = ", show(s), ";\n" ] ;
```

# Presentation Outline

# A Simple Knapsack Problem I

Set Valued Variables – a decision variable can take set as its value.
Example declaration:

```
var set of Items: knapsack;
```

Note: the var keyword comes before the set declaration indicating that the set itself is the decision variable. This contrasts with an array in which the var keyword qualifies the elements in the array rather than the array itself since the basic structure of the array is fixed, i.e. its index set.

**Simple Knapsack Problem**

Given a set of items, find optimal packing of a knapsack, so that:

- weight constraint: the total weight of the selected items is still under the knapsack capability,
- max of value: the total value of the selected elements is maximal.

Note: We do not know the final number of selected elements.
Note: No direct iteration over sets is admissible! The code below will result with an error.

# A Simple Knapsack Problem II

```
constraint sum (i in knapsack) (weights[i]) <= capacity;

solve maximize sum (i in knapsack) (profits[i]) ;
```

# MiniZinc Coding: Knapsack Problem I

```
int: n;
set of int: Items = 1..n;
int: capacity;

array[Items] of int: profits;
array[Items] of int: weights;

% setvar
var set of Items: knapsack;
% capacity
constraint sum (i in Items)
               (bool2int(i in knapsack)*weights[i]) <= capacity;

solve maximize sum (i in Items) (bool2int(i in knapsack)*profits[i]

output [show(knapsack),"\n"];
```

# Example: Social Golfers

## Developing a Schedule for a Tournament

The aim is to schedule a golf tournament over `weeks` using `groups` $\times$ `size` golfers.
For each week we have to schedule `groups` different groups each of size `size`.
No two pairs of golfers should ever play in two groups.
The variables in the model are sets of golfers `Sched[i,j]` for the i-th week and j-th group,

# MiniZinc Coding: Social Golfers I

```
include "partition_set.mzn";
int: weeks;
int: groups;
int: size;
int: ngolfers = groups*size;

array[1..weeks,1..groups] of var set of 1..ngolfers: Sched;

% constraints
constraint
    forall (i in 1..weeks-1) (
                Sched[i,1] < Sched[i+1,1]
    ) /\
    forall (i in 1..weeks, j in 1..groups) (
           card(Sched[i,j]) = size
        /\ forall (k in j+1..groups) (
                Sched[i,j] < Sched[i,k]
```

# MiniZinc Coding: Social Golfers II

```
            /\ Sched[i,j] intersect Sched[i,k] = {}
          )
    ) /\
    forall (i in 1..weeks) (
          partition_set([Sched[i,j] | j in 1..groups], 1..ngolfers)
       /\ forall (j in 1..groups-1) (
              Sched[i,j] < Sched[i,j+1]
          )
    ) /\
    forall (i in 1..weeks-1, j in i+1..weeks) (
        forall (x in 1..groups, y in 1..groups) (
            card(Sched[i,x] intersect Sched[j,y]) <= 1
        )
    );
% symmetry
  constraint
    % Fix the first week %
    forall (i in 1..groups, j in 1..size) (
```

# MiniZinc Coding: Social Golfers III

```
        ((i-1)*size + j) in Sched[1,i]
    ) /\
    % Fix first group of second week %
    forall (i in 1..size) (
        ((i-1)*size + 1) in Sched[2,1]
    ) /\
    % Fix first 'size' players
    forall (w in 2..weeks, p in 1..size) (
        p in Sched[w,p]
    );

solve :: set_search([Sched[i,1] | i in 1..weeks] ++
                    [Sched[i,j] | i in 1..weeks, j in 2..groups
                    input_order, indomain_min, complete)
        satisfy;

output [ show(Sched[i,j]) ++ " " ++
        if j == groups then "\n" else "" endif |
```

# MiniZinc Coding: Social Golfers IV

```
i in 1..weeks, j in 1..groups ];
```

# Presentation Outline

# Cumulative Use of Resources I

The `cumulative` constraint is used for describing cumulative resource usage. Declaration:

```
cumulative(array[int] of var int: s, array[int] of var int: d,
array[int] of var int: r, var int: b)
```

Requires that a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time.

# MiniZinc Coding: The Moving Example I

```
include "cumulative.mzn";

int: n; % number of objects;
set of int: OBJECTS = 1..n;
array[OBJECTS] of int: duration; % duration to move
array[OBJECTS] of int: handlers; % number of handlers required
array[OBJECTS] of int: trolleys; % number of trolleys required

int: available_handlers;
int: available_trolleys;
int: available_time;

array[OBJECTS] of var 0..available_time: start;
var 0..available_time: end;

constraint cumulative(start, duration, handlers, available_handlers
constraint cumulative(start, duration, trolleys, available_trolleys
```

# MiniZinc Coding: The Moving Example II

```
constraint forall(o in OBJECTS)(start[o] +duration[o] <= end);

solve minimize end;

output [ "start = ", show(start), "\nend = ", show(end), "\n"];
```

# MiniZinc Coding: The Moving Data I

```
n = 8;
% piano, fridge, double bed, single bed, wardrobe, chair, chair, ta

duration = [60, 45, 30, 30, 20, 15, 15, 15];
handlers = [3, 2, 2, 1, 2, 1, 1, 2];
trolleys = [2, 1, 2, 2, 2, 0, 0, 1];

available_time = 180;
available_handlers = 4;
available_trolleys = 3;
```