Languages and Algorithms for Artificial Intelligence

Anna Valanzano, Michele Milesi, Pietro Epis

February 2022

Abstract

In this paper we reported a short explanation about the programs that we have developed to solve the Bocconi mathematical games. Each puzzle has been solved in Minizinc and most of them have been developed also in Prolog. In this report we included the listings of the Minizinc code only for the sake of conciseness, anyhow, all the Prolog solution can be found on this GitHub repository, that will be turned public after the project delivery deadline.

1 Exercise 1

```
include "globals.mzn";

int: n_deck_1 = 25;
int: n_deck_2 = 27;
int: n_red_1 = 12;
var int: n_red_2;
var int: n_black_1;
var int: n_black_2;

constraint n_red_1 + n_red_2 + n_black_1 + n_black_2 = n_deck_1 + n_deck_2;
constraint n_red_1 + n_red_2 = (n_deck_1 + n_deck_2) / 2;
constraint n_black_1 + n_black_2 = (n_deck_1 + n_deck_2) / 2;
constraint n_red_1 + n_black_1 = n_deck_1;
constraint n_red_2 + n_black_2 = n_deck_2;

solve satisfy
```

The task was to compute the number of black cards contained in the second deck, assumed that the first one contains 12 red cards. In order to generalize the procedure we defined three parameters: *n_deck_1*, *n_deck_2*, *n_reds_1*, that are respectively the number of cards in the first deck, the number of cards in the second deck and the number of red cards in the first deck.
The first constraint tells that the sum of red and black cards of both decks must be equal to the total number of cards. The second constraint imposes that half of the deck is made up by red cards, and simmetrically the third constraint does the same as regards black cards. The last two constraints state that the number of cards of a deck is equal to the sum of its red and black cards. The result can be read in the variable *n_black_2*.

2 Exercise 2

```
1  include "globals.mzn";
2
3  int: h = 4;
4  int: w = 5;
5  int: unit_distance = 10;
6  array[1..h, 1..w] of 0..1: map = array2d(1..h, 1..w, [
7    0, 1, 1, 1, 0,
8    1, 1, 1, 1, 1,
9    1, 1, 1, 1, 1,
10   0, 1, 1, 1, 0
11 ]);
12 array[1..2] of int: start = [1, 2];
13 array[1..2] of int: end = [3, 5];
14 array[1..(h * w), 1..2] of var int: path;
15 var int: dist;
16 var int: n_steps;
17
18 constraint n_steps >= 0 /\ n_steps <= (h * w);
19 constraint path[1, 1] = start[1] /\ path[1, 2] = start[2];
20 constraint path[n_steps, 1] = end[1] /\ path[n_steps, 2] = end[2];
21 constraint forall(i in 1..n_steps)
22   (path[i, 1] >= 1 /\ path[i, 1] <= h /\ path[i, 2] >= 1 /\ path[i, 2] <= w);
23 constraint forall(i in 2..n_steps)
24   (abs(path[i, 1] - path[i - 1, 1]) + abs(path[i, 2] - path[i - 1, 2]) = 1);
25 constraint forall(i in 1..n_steps)
26   (map[path[i, 1], path[i, 2]] = 1);
27 constraint forall(i in 1..n_steps)
28   (forall(j in 1..n_steps)(i != j -> (path[i, 1] != path[j, 1] \/ path[i, 2] != path[j, 2])));
29 constraint dist = (n_steps - 1) * unit_distance;
30
31 solve maximize dist;
```

The grid shown in the figure has been represented through the matrix *map* (whose dimensions has been parametrized with the variables h and w), where the ones indicate the presence of a red dot and the zeros its absence. An additional parameter *unit_distance* allows to define the distance between two dots (set to 10 as required by the exercise).
The initial and the final dots of the path have been represented in two arrays of two elements (*start* and *end*), each of them storing both the information about the row and the column of the dot.
The variable *path* is the sequence of points that are crossed in order to reach the end point from the start one. It's defined as a two-dimensional array, whose width is equal to the maximum possible number of steps (h * w) and whose height is 2 (each point is described by two coordinates).
The variable *dist* is the most relevant of the program and is indeed the one that is maximized to produce the output. It's defined as the number of steps times the *unit_distance*.

The second and the third constraints impose that first item of the path is the *start* point and the last item of the path is the *end* point. The fourth constraint ensures that the computed coordinates of the points in the path are inside the range defined by the size of the grid. The next one formalizes the constraint that each move can be either a unitary horizontal or vertical step. To be sure that the path doesn't include the points of the grid that are not available, we introduced the fifth constraint, that checks the values in the *map* array. The constraint at line 27 imposes that a point is crossed at most once.

3 Exercise 3

```
include "globals.mzn";
int: h = 5;
int: w = 9;
array[1..h, 1..w] of 0..1: map = array2d(1..h, 1..w, [
  0, 0, 0, 0, 0, 0, 1, 1, 1,
  0, 0, 0, 0, 0, 0, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1,
  0, 0, 0, 0, 1, 1, 1, 1, 0,
  0, 0, 0, 0, 1, 1, 1, 1, 1
]);
array[1..(h * w), 1..2] of var 1..h: rectangles_i;
array[1..(h * w), 1..2] of var 1..w: rectangles_j;

var 1..(h*w): n_rectangles;


constraint forall(i in 1..n_rectangles)(
  rectangles_i[i, 1] <= rectangles_i[i, 2] /\ rectangles_j[i, 1] <= rectangles_j[i, 2]
);
constraint forall(i in 1..h)(
  forall(j in 1..w)(
    map[i, j] = 1 -> exists(k in 1..n_rectangles)(
      rectangles_i[k, 1] <= i /\ i <= rectangles_i[k, 2] /\ rectangles_j[k, 1] <= j /\ j <= rectangles_j[k, 2]
    )
  )
);
constraint forall(i in 1..n_rectangles)(
  forall(j in 1..n_rectangles)(
    i < j -> (rectangles_i[i, 2] < rectangles_i[j, 1] \/ rectangles_j[i, 2] < rectangles_j[j, 1])
  )
);

constraint forall(i in 1..n_rectangles)(
  forall(j in rectangles_i[i, 1]..rectangles_i[i, 2])(
    forall(k in rectangles_j[i, 1]..rectangles_j[i, 2])(
      map[j, k] = 1
    )
  )
);
solve minimize n_rectangles;
```

In order to represent the scheme, we imagined to put a grid over it, to split the figure in squares. Therefore, we stored it as a matrix, in which a one means that the cell is part of the drawing and a zero stands for an empty space. The idea on which the implementation has been based, is about finding the minimum number of rectangles that can be fit into the figure in order to cover its whole surface. Each internal rectangle is described by the top-left and bottom-right corners coordinates, stored in two respective arrays: *rectangles_i* contains the *i-th* coordinate of both points, whereas *rectangles_j* is used to store their *j-th* coordinate (indeed both arrays have 2 as second dimension size). Finally, *n_rectangles* is the target variable to be minimized.

The first constraint states that for each internal rectangle, the coordinates of the top-left corner must be less than or equal to the coordinates of the corresponding bottom-right corner. The second constraint formalizes the concept for which every point of the figure (therefore whose corresponding cell of matrix *map* has value 1) must be contained into one of the rectangles that have been found. The third constraint simply ensures that the internal rectangles don't overlap (assuming an ordering among the rectangles for speed purposes). The last constraint, checks that every rectangle that has been found actually is a rectangle and there are no holes inside of it, so that every point contained between the top-left and the bottom-right corner is part of the drawing.

4 Exercise 4

```
include "globals.mzn";

array [1..4] of var 0..9: digits;
var int: sum;

constraint sum = (
  digits[1] * 1000  +
  digits[2] * 100   +
  digits[3] * 10    +
  digits[4]
);
constraint digits[1] != 0;
constraint forall(i in 1..4)(digits[i] mod 2 = 0);
constraint all_different(digits);
constraint sum mod 11 = 0;

solve minimize sum;
```

In order to implement the constraint related to the fact that every digit of the number must be different, we represented the number as an array (*digits*), in which the *i-th* position contains the *i-th* digit. The relation between the array of digits and the number is obtained through the weighted sum of the digits (in the variable *sum*).

The constraints simply formalize straightforwardly the requirements of the exercise. Since the goal of the exercise is to find the smallest number satisfying the constraints, we minimized the value of *sum*.

5 Exercise 5

```
include "globals.mzn";

int: sum = 20;
array[1..9] of var 1..9: pos;

constraint pos[1] = 1;
constraint pos[2] = 8;
constraint pos[4] = 5;
constraint pos[6] = 4;
constraint pos[9] = 7;
constraint sum(i in 1..4)(pos[i]) = sum;
constraint sum(i in 4..7)(pos[i]) = sum;
constraint sum(i in 7..9)(pos[i]) + pos[1] = sum;
constraint all_different(pos);

solve satisfy
```

The exercise has been parametrized according to the sum of the internal cells of the triangle, defined in the parameter *sum*. The values of the triangle have been represented with an array (*pos*), that reflects the cells of the figure, starting from the top and in clockwise order. The right edge of the triangle is made up by the first four cells, the bottom edge refers to the sum of the *4-th*, *5-th*, *6-th* and *7-th* cells, and finally the left edge is composed of the last three positions of the array along with the first one.

6 Exercise 6

```
include "globals.mzn";

array[1..9] of var 1..9: pos;
var int: l1;
var int: l2;
var int: l3;
var int: sum;

constraint sum(i in 1..4)(pos[i]) = l1;
constraint sum(i in 4..7)(pos[i]) = l2;
constraint sum(i in 7..9)(pos[i]) + pos[1] = l3;
constraint all_different(pos);
constraint sum = l1 + l2 + l3;

solve maximize sum;
```

The representation of the triangle is the same as the one explained in the Exercise 5. The only difference is the introduction of the variable *sum*, defined as the sum of the values of all the edges (and the value of the edge is computed as previously explained). The target is no longer to find a set of suitable values, but to maximize the value of the *sum*.

7 Exercise 7

```
include "globals.mzn";

array [1..11, 1..4] of var 0..9: numbers;
var int: initial;

constraint initial =
  numbers[1, 1] * 1000  +
  numbers[1, 2] * 100   +
  numbers[1, 3] * 10    +
  numbers[1, 4];

constraint forall (i in 1..11) (
  (numbers[i, 1] * 1000 +
  numbers[i, 2] * 100   +
  numbers[i, 3] * 10    +
  numbers[i, 4]) = initial * i
);

constraint forall (i in 1..11) (
  numbers[i, 1] = 9 \/
  numbers[i, 2] = 9 \/
  numbers[i, 3] = 9 \/
  numbers[i, 4] = 9
);

solve satisfy
```

In order to deal easily with the digits of the numbers, each number is represented with a 4-elements array. All the numbers of the sequence (the 11 multiples) are grouped into a unique array (*numbers*), whose size is therefore 11 x 4. The *i-th* row contains the digits of the *i-th* multiple of

the intial number.

The matching between the *i-th* row and the corresponding number is formalized through the second constraint. Since the goal is to discover which was the initial number, this is defined in the variable *initial*, that is then constrained to be equal to the weighted sum of the elements of the first row of the array. The last constraint imposes that at least one digit of each multiple is equal to 9 (as required by the exercise).

8 Exercise 8

```
include "globals.mzn";

int: birth_year = 2000;
var 0..9: c1;
var 0..9: c2;
var 0..9: c3;
var 0..9: c4;
var int: sum;

constraint sum = c1 + c2 + c3 + c4;
constraint sum = ((c1 * 1000 + c2 * 100 + c3 * 10 + c4) - birth_year)

solve satisfy
```

We used four variables, one for each digit of the number representing the year, so that the variable c_i contains the value of the *i-th* digit.

The two constraints are trivial, and simply formalize the requirements of the exercise.

9 Exercise 9

```
include "globals.mzn";

array[1..8] of var 0..9: date;

int: start_date = 20210919;

constraint date[7] <= 3;
constraint date[7] == 3 -> (date[8] == 0 \/ date[8] == 1);
constraint date[5] <= 1;
constraint date[5] == 1 -> date[6] <= 2;
constraint forall(i in 1..8)(count(date, date[i], 2));
constraint nvalue(4, date);
constraint sum(i in 1..8)(10^(8 - i) * date[i]) > start_date;

solve minimize sum(i in 1..8)(10^(8 - i) * date[i]);
```

Since a date is made up by 8 digits (2 for the day, 2 for the month and the last 4 for the year), it's been represented with an array of 8 elements. To achieve an ordering, we found it more convenient to invert the parts of the date, and so we put first the year, followed by the month and the day. The initial date has been expressed as a number, obtained by the weighted sum of its digits.

The first four constraints simply force the array to represent a semantically valid date. The fifth constraint states that each digit is present only once in the array. This is checked by taking advantage of the function *count/3*. The next constraint makes use of the MiniZinc function *nvalue/2*, that tests if the first parameter is equal to the number of distinct values inside an array. The last constraint simply ensures that the date produced as output follows the initial one.

10 Exercise 10

```
include "globals.mzn";

int: perimeter;
var int: l1;
var int: l2;
var int: gray_tiles;

constraint l1 > 0 /\ l2 > 0;
constraint 2 * l1 + 2 * l2 - 4 = perimeter;
constraint gray_tiles = (l1 * l2) - perimeter;

solve maximize gray_tiles;
```

In order to generalize the problem, the number of tiles of which the perimeter is made up by is represented as parameter. Note that the gray tiles can be computed as the area of the rectangle minus the perimeter. The variables of the problem are the length of the two edges of the rectangle (*l1* and *l2*) and the number of gray tiles (*gray_tiles*).

In the second constraint, whose target is to express the perimeter as a function of the edges, we subtract 4 because otherwise the corners would be considered twice into the sum. The last constraint defines the internal area, and therefore the number of gray tiles, in the previously explained way.

11 Exercise 11

```
include "globals.mzn";

array[1..4] of var 0..9: digits;
var int: number;

constraint number =
  digits[1] * 1000  +
  digits[2] * 100   +
  digits[3] * 10    +
  digits[4];
constraint digits[1] != 0;
constraint exists(i in 1..4)(digits[i] = 0);
constraint (digits[2] = 0 /\ (digits[1] * 100   +
  digits[3] * 10    +
  digits[4]) = number / 9) \/
  (digits[3] = 0 /\ (digits[1] * 100   +
  digits[2] * 10    +
  digits[4]) = number / 9) \/
  (digits[4] = 0 /\ (digits[1] * 100   +
  digits[2] * 10    +
  digits[3]) = number / 9);

solve minimize number;
```

The task was to find the smallest 4 digits number such that if we delete a 0 we obtain a ninth of the number itself.
Also in this case we represented the number as a 4 dimensional array and we used a constraint

to prevent the first digit from being 0, because otherwise the number of digits would be only 3. The other two constraints impose that at least a digit is 0 and they take in consideration the three possible positions of the digit 0: for each possible position they require the weighted sum of the other three digits to be the ninth of the original number.

12 Exercise 12

```
include "globals.mzn";

var 0..9: c1;
var 0..9: c2;

constraint (c1 * 10 + c2) * 4 - 3 = (c2 * 10 + c1);
```

In this exercise we represent a 2 digits number with two variables c_1 and c_2 and we used a simple constraint to impose that if we multiply the number by 4 and we subtract 3 the resulting number has the same digits of the original number, but in different order.

13 Exercise 13

```
include "globals.mzn";

int: c1;
int: c2;
var int: n;

constraint n * (c1 ^ 3) <= (c2 ^ 3);

solve maximize n;
```

We used two variables c_1 and c_2 to represent the edges of two cubes and a variable n that represent the number of times that the greatest cube can contain the smallest one. We maximized n. The biggest edge is c_2 and c_1 is then the smallest one.

14 Exercise 14

```
include "globals.mzn";

int: n_faces = 6;
int: n_vertices = 8;
int: n_initial_cuts;
var int: n_initial_faces;
var int: n_initial_vertices;
var int: n_faces_after_cut;

constraint n_initial_faces = n_faces + n_initial_cuts;
constraint n_initial_vertices = n_vertices + 2 * n_initial_cuts;
constraint n_faces_after_cut = n_initial_faces + n_initial_vertices;
constraint n_initial_cuts <= n_vertices /\ n_initial_cuts >= 0;

solve satisfy;
```

We have a cube, and we make a cut around one of its vertices. The task was to find the number of faces of the solid if we perform a cut on each of its vertices. To express the relation between the number of faces and the number of vertices, we found out that every cut around a vertex introduces an additional face and two new vertices.

We pursued the goal of making the problem as parametrized as possible, indeed we introduced a parameter *n_initial_cuts* that allows to change the number of cuts that have been performed on the cube before computing the main task.

For these reasons, the first constraint computes the number of faces at the beginning of the execution as the sum of the faces of the solid (without any cuts) and the number of cuts already performed (since every cut introduces a new face). The second constraint is related to the same concept, but as regards the number of vertices. Since the final number of faces depends on the initial number of faces and the initial number of corners, we define it as their sum, in the variable *n_faces_after_cut*.

15 Exercise 15

```
include "globals.mzn";

int: path_width;
int: area;
var int: actual_area;
var int: b;
var int: h;

constraint b > 10;
constraint h > 10;
constraint b > h;
constraint b * h = area;
constraint actual_area = area - path_width * h;

solve satisfy;
```

In this exercise we have to find the area of a portion of a rectangle. Our intuition is that the area of the path can be computed as it was a parallelogram. We use the variable *path_width* to store the length of the base of the path and two variables b and h to store the parallelogram dimensions. We used a simple constraint to impose that the remaining area is the area of the total rectangle minus the area of the path.

16 Exercise 16

```
include "globals.mzn";

array[1..6] of var 0..9: time;

constraint time[1] == 2 -> time[2] < 4;
constraint time[1] <= 2 /\ time[3] < 6 /\ time[5] < 6;
constraint all_different(time);
constraint sum(time) = 15;

solve satisfy;

% To get all solutions set geocode options
% Configuration Editor -> Solving (User defined behavior) -> Uncheck "Stop after this many solutions"
% Check "Output solving statistics"
```

The task was to find the number of times that an alarm indicates simultaneously the numbers 0,1,2,3,4,5. We imposed some constraints to construct a realistic hour, for example imposing that

the first digit cannot be higher than 2. Then we imposed that the sum of all the digits must be 15, that is the sum of 0,1,2,3,4,5 and that they must be all different.

17 Exercise 17

```
include "globals.mzn";

var int: N;
var int: N3;
var int: N4;
int: max_digits = 9;
array[1..max_digits] of var 0..9: N3_digits;
array[1..max_digits] of var 0..9: N4_digits;

constraint sum(i in 1..max_digits)(10^(max_digits - i) * N3_digits[i]) = N3;
constraint sum(i in 1..max_digits)(10^(max_digits - i) * N4_digits[i]) = N4;
constraint N3 = N^3;
constraint N4 = N^4;
constraint alldifferent_except_0(N3_digits ++ N4_digits);
constraint forall(i in 0..9)(exists(j in 1..max_digits)(N3_digits[j] = i \/ N4_digits[j] = i));

solve satisfy;
```

The task is to find an integer and positive number N such that we have to use all the ten digits to write N^3 and N^4. We used three integer variables to store the values of N, N_3 and N_4 and the digits of N^3 and N^4 are stored in two arrays *N3_digits* and *N4_digits*. The first constraints impose that the weighted sum of *N3_digits* and *N4_digits* are respectively equal to the numbers *N3* and *N4*. The other constraints impose that the digits in *N3_digits* and *N4_digits* must be all different except for the zero digits, because the two arrays contain 10 elements, therefore there will be more than one element equal to 0. The last constraint imposes that each digit has to appear at least once either in *N3_digits* or in *N4_digits*.

18 Exercise 18

```
include "globals.mzn";

var 0..11: h;
var 0..59: m;
var 0..59: s;
var 0..360: h_degree;
var 0..360: m_degree;
var 0..360: s_degree;

constraint h_degree = (h * 30);
constraint m_degree = (m * 6);
constraint s_degree = (s * 6);
constraint abs(h_degree - m_degree) = 120;
constraint abs(h_degree - s_degree) = 120;
constraint abs(m_degree - s_degree) = 120;

solve satisfy;
```

The task is to find the number of times in which the clock hands form an equilateral triangle. We used three variables to store the hours, the minutes and the seconds and other three variables to store the angles that the three clock hands form with the vertical axis.

We used three simple constraints to impose the differences between the angles to be 120, because if the resulting triangle is equilateral the corners in the center must be all equal ($360 : 3 = 120$).