

Relation for LAAI Project

Daniele Baiocco, matr. 0001057671

February 17, 2022

Contents

1	Introduction	2
2	Bocconi's games	2
2.1	Cards	2
2.2	The smallest	4
2.3	Magic triangle	5
2.4	Maximum triangle	6
2.5	How many nines	7
2.6	Matteo's age	8
3	Coding game puzzle	9
3.1	Nurikabe	9

1 Introduction

The project group is composed only by myself **Daniele Baiocco**, matr. 0001057671, email: daniele.baiocco@studio.unibo.it.

For this project I've used Minizinc to solve all the problems I addressed. I also parameterized the behavior of all the programs: It is possible to change the input values contained in the data file (.dzn) of each Minizinc program in order to get a different solution. The validity of the inputs is checked via various **asserts**. The input data is always instantiated with the values contained in the problems.

All the source code is available at this [github repository](#).

2 Bocconi's games

I've chosen to solve 6 of the 18 puzzle exercises taken from Bocconi's mathematical games that you can find [here](#). The correctness of the outputs can be examined by comparing them with the actual solutions contained [here](#). The games are **Rosso e nero** (numero 1), **Il più piccolo** (numero 4), **Un triangolo magico** (numero 5), **Un triangolo massimo** (numero 6), **Quanti 9!** (numero 7), **L'età di Matteo** (numero 8). The names of the problems have been translated in english in the following sections.

2.1 Cards

The **input parameters** are:

- **n_cards**: total number of cards in the deck,
- **n_red**: number of red cards,
- **left_split**: number of cards in the first split of the two.

The number of **black cards** and the number of **cards in the second split** are derived by these three input values by doing respectively **n_cards - n_red** and **n_cards - left_split**.

The **variables** are:

- **n_red_left_split**: the number of red cards in the first split, which has his domain in the range $0..n_red$,
- **n_black_left_split**: the number of black cards in the first split, which has his domain in the range $0..n_black$,
- **n_red_right_split**: the number of red cards in the second split, which has his domain in the range $0..n_red$,
- **n_black_right_split**: the number of black cards in the second split, which has his domain in the range $0..n_black$.

The **constraints** I implemented are quite trivial:

- the number of black cards in the second split must be equal to the number of black cards minus the number of black cards in the first split:

```
n_black_right_split=n_black-n_black_left_split;
```

- the number of red cards in the second split must be equal to the number of red cards minus the number of red cards in the first split:

```
n_red_right_split=n_red-n_red_left_split;
```

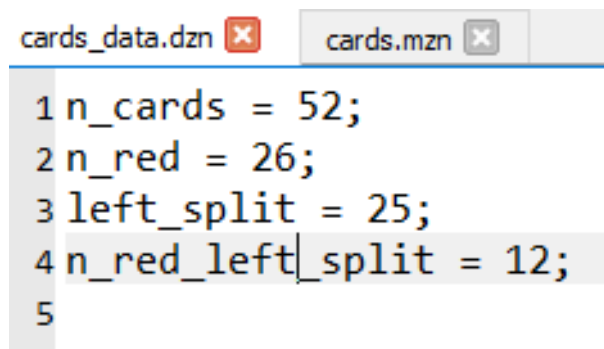
- the number of cards in the first split must be equal to the number of black cards plus the number of red cards both in the first split:

```
left_split=n_red_left_split+n_black_left_split;
```

- the number of cards in the second split must be equal to the number of black cards plus the number of red cards both in the second split:

```
right_split=n_red_right_split+n_black_right_split;
```

Putting four variables allows to have a program not only capable of solving the problem with the initial configuration, but also with different other ones: I could specify a value in the .dzn file for the number of red cards in the second split, along with values for the three inputs in order to have in output the number of black cards in the first split, the number of black cards in the second and the number of red cards in the first split, or I could specify a value for the number of black cards in the second split in order to get the other three variables and so on. In Figure 1 there are the inputs taken from the Bocconi exercise and in Figure 2 there is the output of the computation.



```
cards_data.dzn × cards.mzn ×  
1 n_cards = 52;  
2 n_red = 26;  
3 left_split = 25;  
4 n_red_left_split = 12;  
5
```

Figure 1: Input values from the "Cards" exercise

```
Output
Running cards.mzn with cards_data.dzn
The card deck contains 12 red cards and 13 black cards in the first splitted deck.
It contains 14 red cards and 13 black cards in the second one.
-----
Finished in 169msec
```

Figure 2: Output from the "Cards" exercise

2.2 The smallest

The **input parameters** are:

- **num_digits**: number of digits of the number the solver needs to find,
- **multiplier**: the number the output variable needs to be multiple of.

From **num_digits** I computed the lower bound and the upper bound which defines the **domain range** of the output number by doing respectively

```
int: lower_bound=10^(num_digits-1);
```

and

```
int: upper_bound=10^(num_digits)-1;
```

I also computed an array of powers of 10 called **powers** in order to have a way to retrieve each digit of the output variable.

The only **variable** is **result** which may take, as already said, a value in the range **lower_bound..upper_bound**.

From this variable and the power array I computed a list of **var int** which contains each digit of **result** by doing:

```
list of var int: digits=[(result div n) mod 10 | n in powers];
```

This list is helpfull in the formalization of the constraints.

The constraints I defined are the following:

- all the digits in **result** must be different:

```
alldifferent(digits);
```
- every digit in **result** must be even:

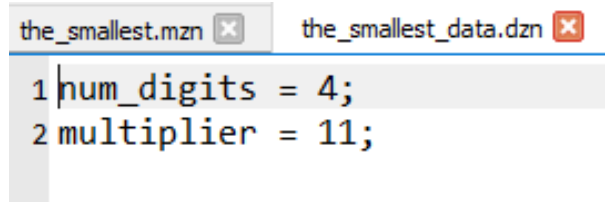
```
forall(digit in digits)(digit mod 2==0);
```
- **result** must be a multiple of **multiplier**:

```
result mod multiplier==0;
```

- **result** must be the smallest solution the solver can find:

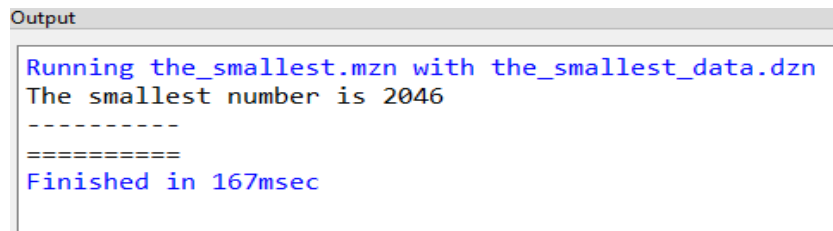
```
solve minimize result;
```

In Figure 3 there are the inputs taken from the Bocconi exercise and in Figure 4 there is the output of the computation.



```
the_smallest.mzn
1 | num_digits = 4;
2 | multiplier = 11;
```

Figure 3: Input values from the "The smallest" exercise



```
Output
Running the_smallest.mzn with the_smallest_data.dzn
The smallest number is 2046
-----
=====
Finished in 167msec
```

Figure 4: Output result from the "The smallest" exercise

2.3 Magic triangle

The **input parameter** is **starting_conf** which is a list of digits from 0 to 9 of length 9. There is only one **variable** called **result** which is a list of digits from 1 to 9 of length 9. They both represent a triangle, in which elements in position 1, 2, 3, 4 represent the first side of the triangle, elements in 4, 5, 6, 7 the second side and elements in 1, 7, 8, 9 the third. The elements in position 1, 4 and 7 are the vertex of the triangle.

The **constraints** are:

- for each digit in the starting_conf, if it is different from 0 then this digit must be in the **result** array in the same position:

```
forall(i in 1..9)(
  if starting_conf[i] > 0
  then result[i] = starting_conf[i] else true endif
);
```

- the sum of the digits in the first side of the triangle must be 20:

```
result[1] + result[2] + result[3] + result[4] = 20;
```

- the sum of the digits in the second side must be 20:

```
result[4] + result[5] + result[6] + result[7] = 20;
```

- the sum of the digits in the third side must be 20:

```
result[7] + result[8] + result[9] + result[1] = 20;
```

- each digit must be unique:

```
alldifferent(result);
```

In Figure 5 there are the inputs taken from the Bocconi exercise while in Figure 6 there is the output of the computation. From the output we can say that Debora wrote in the the bottom-left vertex the number 9.

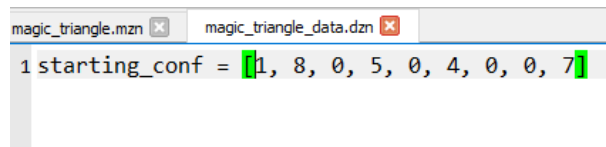


Figure 5: Input values from the "Magic trinagle" exercise

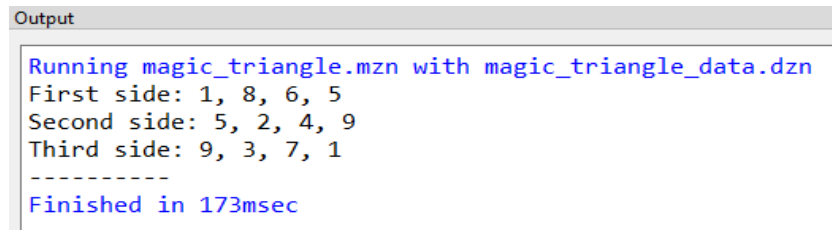


Figure 6: Output result from the "Magic triangle" exercise

2.4 Maximum triangle

This exercise is similar to the previous one. There is only one **variable** called **result** which is, as before, an array containing digits from 1 to 9 of length 9. There isn't a .dzn file, because the number of the array must be equal to the number of digits and it's not parametrizable.

The **constraints** are:

- digits must be all different:

```
alldifferent(result);
```

- **result** must be instantiated with digits in a way that maximises the sum of each side of the triangle:
`solve maximize first_side + second_side + third_side;`
 with `first_side`, `second_side`, `third_side` computed by adding the indexes of **result** specified in the initial part of the **Magic triangle** puzzle game of this article.

In Figure 7 there is the output of the computation.

```

Output
Running maximum_triangle.mzn
The maximum sum is 69and it is achieved
through this configuration: [7, 2, 1, 9, 5, 4, 8, 6, 3]
-----
=====
Finished in 185msec

```

Figure 7: Output from the "Maximum triangle" exercise

2.5 How many nines

The **input parameters** are:

- **n_digits**: the number of digits that the output number must have,
- **max_digits_after_mult**: the number of digits of the number obtained multiplying the output number with the greater value in the `multiplicator_range`,
- **multiplicator_range**: each number in this range represents the value that must be multiplied with the output number,
- **num_to_check**: it represents the number that must be in at least one digit of each multiplication between the output number and the values in the `multiplicator_range`.

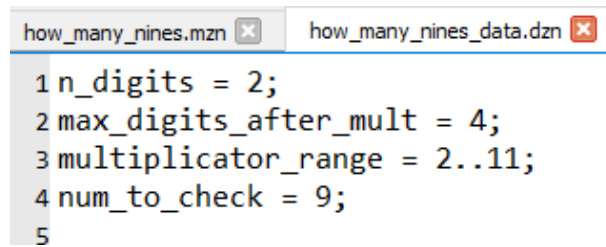
There is one single **variable** called **result** and one single **constraint**. This constraint tells that for each number in the `multiplicator_range`, if it is multiplied with the result to return, there exist at least a digit that is equal to the `num_to_check` and it's expressed in this way:

```

forall(i in multiplicator_range)(
    exists(n in powers)(
        ((result * i) div n) mod 10 == num_to_check
    )
);

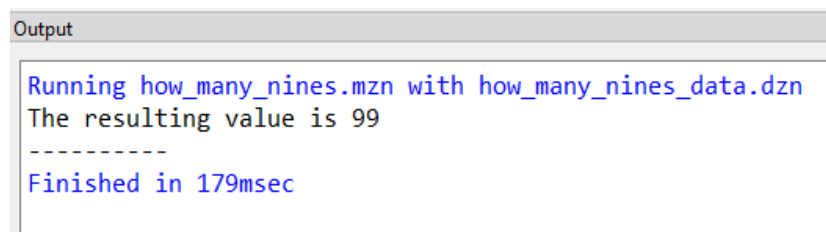
```

In Figure 8 there are the inputs taken from the Bocconi exercise while in Figure 9 there is the output of the computation.



```
1 n_digits = 2;
2 max_digits_after_mult = 4;
3 multiplicator_range = 2..11;
4 num_to_check = 9;
5
```

Figure 8: Input values from the "How many nines" exercise



```
Output
Running how_many_nines.mzn with how_many_nines_data.dzn
The resulting value is 99
-----
Finished in 179msec
```

Figure 9: Output result from the "How many nines" exercise

2.6 Matteo's age

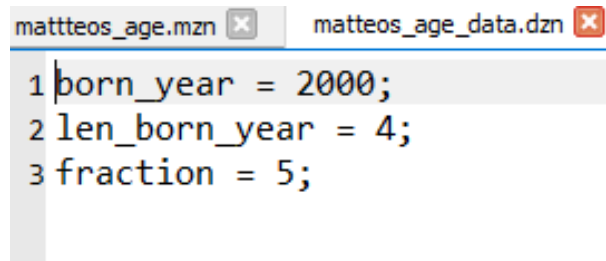
The **input parameters** are:

- **born_year**: the number which represents the born date,
- **len_born_year**: the number of digits of born_year,
- **fraction**: the number used in the fraction constraint.

The only **variable** I defined is target_year, while the **constraints** have been implemented as follows:

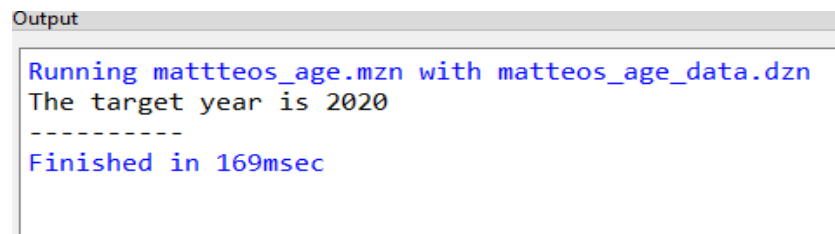
- the sum of each digit in the target year must be equal to the difference between the born year and the target year divided by the parameter fraction:
`sum(digits) == (target_year - born_year) div fraction;`
- the target year must be greater than the born year:
`target_year > born_year;`
- the division with the input value fraction must have rest 0:
`(target_year - born_year) mod fraction == 0;`

In Figure 10 there are the inputs taken from the Bocconi exercise while in Figure 11 there is the output of the computation.



```
matteos_age.mzn x matteos_age_data.dzn x
1 born_year = 2000;
2 len_born_year = 4;
3 fraction = 5;
```

Figure 10: Input values from the "Matteo's age" exercise



```
Output
Running matteos_age.mzn with matteos_age_data.dzn
The target year is 2020
-----
Finished in 169msec
```

Figure 11: Output result from the "Matteo's age" exercise

3 Coding game puzzle

I've also implemented a solution to a gaming puzzle different from the one proposed in the project instructions pdf, but of the same level of difficulty, called **nurikabe**. All the specifications of the problem are listed at [this link](#).

There are two things to take into account before digging into the details of the implementation:

- the solver configuration I've used in the minizinc ide is Chuffed 0.10.4, because the default solver Geocode 6.3.0 doesn't terminate the execution,
- It is used the fact that an access in the grid with indexes out of bounds returns a false evaluation: the compiler implements the policy **undefined result becomes false in Boolean context**.

3.1 Nurikabe

There are two **input parameters**:

- **N**: the dimension of the grid,
- **input_grid**: the grid in input, which contains clues for the islands, in the form of digits in domain 1 to 9, and zeros which represent the cells that will be filled with the right values by the program.

There is only one **variable** called **output_grid**, in which the zeros represent the sea and every group of digits an island. The **constraints** and their implementations are explained as follows:

- the digits that represent the clues in the input_grid must be in the present in the same positions in the output_grid:

```
constraint forall(i in 1..N, j in 1..N)(
  if (input_grid[i,j] != 0)
  then output_grid[i,j] = input_grid[i,j]
  else true endif
);
```

- the number of cells in each island must be equal to the value of the clues taken from the input_grid:

```
forall(i in range_one_r)(
  sum(j in 1..N, k in 1..N where output_grid[j,k] == i)
  (output_grid[j,k]) == i*i
);
```

In the snippet above range_one_r is an array of all the digits that are in the input grid, excluding 0.

- there cannot be 2x2 areas of zeros or more:

```
forall(j in 1..N-1, k in 1..N-1)(
  if(output_grid[j,k] == 0 /\ output_grid[j,k+1] == 0)
  then output_grid[j+1,k]+output_grid[j+1,k+1] != 0
  else true endif
);
```

Here everytime two consecutive cells have value 0, there is the constraint that the two cells under them cannot be both zero.

- each cell must be instantiated by the solver with a digit that appear in the input_grid:

```
forall(j in 1..N, k in 1..N)(
  exists(d in range_zero_r)(output_grid[j,k]==d)
);
```

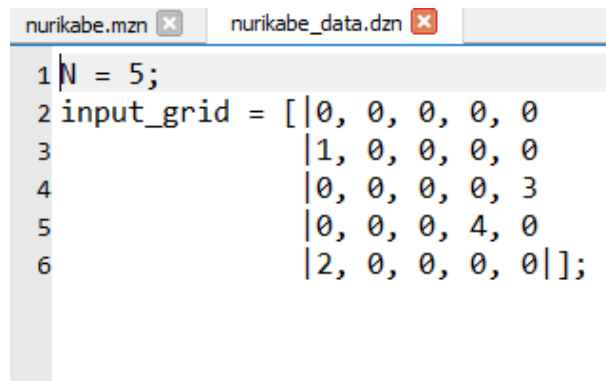
- each island must be isolated from the others horizontally and vertically:

I've implemented a constraint that for each digit, for each cell that has as value that digit, it ensure that the cells around contain either the same digit or 0 (which is a sea cell),

- each island, and also the sea, must have a continuous shape:

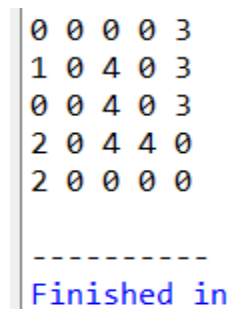
There are two constraints, in conjunction, inside the same forall. For each digit, for each cell that has as value that digit, the first constraint ensures that at least one of the cells around that cell has as value the same digit. The second one ensures that if there is only one single cell x of the same digit, then there must be at least one cell that is in the proximity of x of that digit (different from the starting cell).

In the last two points there aren't lines of code because they would have taken too much space. In Figure 12 there are the inputs taken from the web page while in Figure 13 there is the output of the computation.



```
nurikabe.mzn x nurikabe_data.dzn x
1 N = 5;
2 input_grid = [|0, 0, 0, 0, 0
3               |1, 0, 0, 0, 0
4               |0, 0, 0, 0, 3
5               |0, 0, 0, 4, 0
6               |2, 0, 0, 0, 0|];
```

Figure 12: Input values from the "Nurikabe" exercise



```
0 0 0 0 3
1 0 4 0 3
0 0 4 0 3
2 0 4 4 0
2 0 0 0 0

-----
Finished in
```

Figure 13: Output result from the "Nurikabe" exercise