Report - LAAI Project

Daniele Morotti, Edoardo Procino, Andrea Valente

Academic Year 2021-2022

1 Introduction

In order to learn as much as possibile and to challenge ourselves, we have divided our project in two parts.

In the first part (2) we solved the puzzle game *Nurikabe* [1] using MiniZinc, while, in the second part (3), we solved two exercises of *Bocconi's training for autumn games* [2]. In particular, in the second part, we have solved the exercise 2 only using MiniZinc and the exercise 17 using both MiniZinc and Prolog in order to verify which method is the best one for this kind of problems in terms of execution time and intuitiveness of the solution.

2 Nurikabe

Nurikabe is a Japanese number logic game (NP complete problem) in which at the beginning there is a squared grid representing a group of islands. The goal is to find the shape of each island and fill the rest with water. Clues are given in the form of numbers, each number belongs to a distinct island and tells you the size of its surface. Diagonals do not count as links, so each island must contain the right number of continuous cells. Also the water's cells must be continuous (diagonals do not count) and also there can not be 2x2 or bigger square of water in the final solution. Furthermore, each island must be separated from any other island.

2.1 Nurikabe with different islands

Given the difficulty of the problem, we started by making a further assumption considering all input islands as if they have a different number of cells.

The codification of the problem is very simple and intuitive.
First of all, the user must insert some parameters:

- `length`: the number of cells of the side of the game board;
- `init`: a $length \times length$ matrix which represents the initial game board with clues and zeros that represent empty cells.

[1]https://www.codingame.com/training/expert/nurikabe
[2]https://giochimatematici.unibocconi.it/images/autunno/2021/practiceq.pdf

Finally, we have a variable called `solution` that is a $length \times length$ matrix.
This matrix will be the final solution and we used the following codification for islands and water:

- Islands are represented with numbers greater than 0, in particular, each island is composed by a single number equal to its dimension.
- Water is represented by the number 0;

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 4 & 0 \\ 2 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 & 0 & 3 \\ 1 & 0 & 4 & 0 & 3 \\ 0 & 0 & 4 & 0 & 3 \\ 2 & 0 & 4 & 4 & 0 \\ 2 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Init matrix of a 5x5 game board   Corresponding solution matrix

2.1.1 Rule 1

All clues must be distinct
Given our initial assumption we need to set this constraint.

```
predicate clue_unique(int: val) =
  sum(k in 1..length) (count(init[k,..], val)) == 1;

constraint forall(i,j in 1..length where init[i,j]>0) (
    assert(clue_unique(init[i,j]), "init␣is␣not␣valid"));
```

2.1.2 Rule 2

The number of cells in each island equals the value of the clue.

To achieve this constraint we impose that for all cells in `init` with a value greater than 0, the value of the cell must be equal to the number of cells in the solution matrix with the same value. In other words, if we have a 5 in the `init` matrix, then we want 5 fives in the `solution` matrix.

```
function var int: count_cells(int: val) =
  sum(k in 1..length) (count(solution[k,..], val));

predicate island_count_cells() =
  forall(i,j in 1..length where init[i,j] > 0)(
    count_cells(init[i,j]) == init[i,j]
  );
```

2.1.3 Rule 3

Clues can range from 1 to a maximum of 9.

This is a domain constraint for pruning the search space.

```
    array[1..length,1..length] of var 0..max(init): solution;
```

2.1.4 Rule 4

All islands are isolated from each other horizontally and vertically.

Thanks to our assumption, for this property, we can simply bind all the island-cells in the solution to be surrounded only by cells with the same value or by water cells. Without the assumption, if two or more islands have the same dimension they will merge into a single bigger one and this is not allowed.
This property is symmetric, for this reason the check is done only for values greater than 1.

```
predicate check_neighborhood(int: i,int: j) =
  forall (di,dj in -1..1 where k_adjacency(1,i,j,di,dj)) (
    solution[i+di,j+dj] == solution[i,j] \/ solution[i+di,j+dj] ==
        0
  );
predicate island_isolated() =
  forall(i,j in 1..length where solution[i,j] > 1)
    (check_neighborhood(i,j));
```

2.1.5 Rule 5

There are no water areas of 2x2 or larger.

To achieve this, we check that for all cells — excluded those in the last column and in the last row — the sum of the values of the cells that form the 2x2 square with the examined cell in the top-left corner must be greater than 0.

```
predicate water_not_squares() =
  forall(i,j in 1..length-1)(
    solution[i,j] + solution[i+1,j] +
    solution[i+1,j+1] + solution[i,j+1] > 0
  );
```

2.1.6 Rule 6

When completed, all water and islands cells form a continuous shape

In order to constrain the solution to have a continuous surface, which can be either of island or water, various techniques to solve the puzzle have been implemented throughout the entire development of the project.

In a first time the approach was to put together this rule with another one (2.1.2), this approach was substantially avoided when the implementation was too difficult, for this reason we decided to follow a divide-et-impera approach and consider the two rules as independent from each other.

The core of the rule is the predicate *surface_continuous*. The idea is to check if a cell, which can be either water or island, reaches the cells of the same type by only moving to the adjacent ones with the same value.

Many problems might arise with this predicate, for this reason we have tested some parameters more than the ones that were needed in advance, in order to improve also the efficiency.

The *visited* parameter is a set of the already visited nodes in the path, in this way there is a guarantee that no loops will occur. We choose the set over the array for the ease of checking if a cells wasn't already visited, with the *in* operator, and to add it afterwards with the *union* operator. This choice prevents us to use the conventional way to represent cells, namely *(i,j)*, instead we encode a point in an univocal way with the function *encode_point*.

```
predicate surface_continuous(int: i, int: j, var set of int:
   visited, int: u, int: v, int: res) =
  res > 0 /\ (
    (i == u /\ j == v) \/
    exists (di,dj in -1..1 where k_adjacency(1,i,j,di,dj) /\ not (
       encode_point(i+di, j+dj) in visited)
            /\ solution[u,v] == solution[i+di,j+dj]) (
      (i+di == u /\ j+dj == v) \/ surface_continuous(i+di, j+dj, {
         encode_point(i,j)} union visited, u, v, res-1)
    )
  );
```

Then, the other parameter useful to improve the efficiency is *res*. In fact, during the execution of the model the stack overflow error occurs, for this reason we decided to limit the length of the path to a specific upper bound, which for an island is the size of that and for the water is *water_limit*.

```
function int: all_island_cells() =
  sum (i,j in 1..length where init[i,j] > 0) (init[i,j]);

int: water_cells = length * length - all_island_cells();
int: water_limit = water_cells div 2;
```

2.2 Original Nurikabe

The original nurikabe puzzle, where distinct islands of the same size might occur, cannot be solved in MiniZinc along with the constraints explained in 2.1. The reason is that the input map needs all the clues to be distinct, otherwise it will fail, and there is no way to compute a new input map at run-time in order to satisfy this condition. The only solution that comes to our mind is having it explicitly written by the user, but this would require rewriting all the tests to be performed.
As explained in the official documentation of MiniZinc [3], it is possible to solve a model using the Minizinc Python package, for this reason we decided to develop a simple Python script which performs the execution of the model, that is an analogous task as the MiniZinc IDE, and the data input preprocessing. The latter allows to manage various tests we have available, also ones including islands of the same size.

The new input map is encoded as follows:

- Zeros remains the same;
- Island clues, which range from 1 to 9, are converted into two digits: the leftmost represents the island size, while the rightmost is necessary to represent uniquely islands of the same size.

[3]https://www.minizinc.org/doc-2.5.5/en/python.html

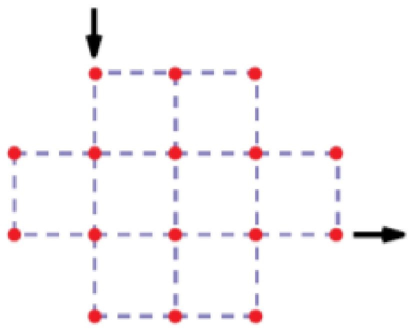
2.3 Insight on Performance

We have made a lot of attempts to improve the efficiency of our model but the solver runs indefinitely with matrices that are larger than 5x5. This may be due to the use of MiniZinc that is not a suitable choice for this problem.

3 Bocconi Exercises

3.1 Exercise 2 - The labyrinth

Given the following scheme we have to find the most expensive path the user can take to get to the final node, starting from the first node and passing only once for each node. Any connection between nodes costs 10.



At the beginning we have considered an encoding for the representation of the nodes and to be effective we have chosen to represent each node with a number, from 1 to 16. With this encoding the node from which the user starts is '1', he/she has to arrive to the node '13' and the solution is an array with the ordered number of the crossed nodes.

Once we have created a 2D array that contains all the connections between the nodes, as if they are edges of a graph, we can start our search.

3.1.1 Rule 1

Since we represent the solution as an array of numbers, we need that all the elements after the target node are equal to zero in order to stop the search and each node before the target node must be greater than zero, otherwise we would not have a path from the start node to the target one.

```
predicate last_not_0(array[int] of var int: x, var int: y) =
  exists(i in 1..length(x)) (
    x[i] = y /\ forall(j in i+1..n) (x[j] = 0) /\
    forall(j in 1..i) (x[j] > 0)
  );
```

3.1.2 Rule 2

With this rule we simply check that a node in the solution is equal to 0 (not in the path) or it has a connection with another node in the 2D matrix of edges.

```
    predicate all_paths(array[int] of var int: x, array[int, 1..2]
        of var int: graph) =
      forall(i in 2..length(x)) (
        x[i] = 0 \/
        (
          x[i] > 0 /\ x[i-1] > 0 /\
          exists(j in 1..num_edges) (
            (graph[j,1] = x[i-1] /\ graph[j,2] = x[i])
            \/
            (graph[j,1] = x[i] /\ graph[j,2] = x[i-1])
          )
        )
      );
```

3.1.3 Rule 3

In this last predicate we call the 2 previous rules and we achieve the 2 remaining goals: passing through each node only once and starting from the first node.

```
predicate compute_solution() =
  alldifferent_except_0(x) /\ x[1] = start /\
    last_not_0(x, target) /\ all_paths(x, graph);
```

To get the most expensive path we simply maximize the following variable:

```
var int: path_length = sum(i in 1..n) (x[i] > 0);
```

3.2 Exercise 17

The goal of this exercise is to find a positive integer N such that all the digits from 0 to 9 are used to write N^3 and N^4. We can consider two possibilities:

1. All digits must appear **only** once.
2. All digits must appear **at least** once.

However the problem is not more complex because we can simply remove or change a simple condition of uniqueness in both MiniZinc and Prolog.

3.2.1 Minizinc implementation

The constraint for checking whether the digit occurs in the first power or the second one is *check_occurence*. In this predicate, the first constraint is satisfied if the j-*th* digit of the number `num` is the one we are searching, that is the parameter i. The second constraint is necessary to avoid false positives of the zero digit, which may happen when we have examined the whole number, otherwise the first constraint will return true.

This predicate is called with both power numbers and a xor/or is added between them. The choice between the two depends on the problem as we have written in the introduction (3.2).

```
    predicate check_occurence(int: i, var int: num) =
      exists (j in 1..digit_limit)
        (i == (num div pow(10, j-1) mod 10) /\
        /\ (num div pow(10, j-1) > 0));
```

Afterwards, we need another constraint to avoid integer overflow. In a first time we have added the constraint as shown in Listing 1, because the fourth power of `N` must be at most $2^{31} - 1$.

```
constraint N > 0 /\ int2float(N) <= sqrt(sqrt(2147483647));
```

Listing 1: Ex 17 - Integer Overflow constraint

We thought that with this constraint the model lost intuitiveness, but its mandatory when the *or* connective is used. Otherwise, when the *xor* is used we have thought about a better constraint, shown in Listing 2.

```
constraint card(0..9) == digits(pow(N, 3)) + digits(pow(N, 4));
```

Listing 2: Ex 17 - Integer Overflow more efficiently

The reasoning on top of this is that both power numbers merged together are a permutation of the set from 0 to 9, in this way the sum of the number of their digits must be equal to the cardinality of the set. This constraint also improves the efficiency in the search space, because a trivial proof shows that only the numbers between 18 and 21 are admissible.

3.2.2 Prolog implementation

Using Prolog with the *clpdf* module we need to consider a different and intricate approach in solving this problem. First of all we need a function that, given a number, returns a list of digits such that we can perform further analysis on it.
Then we create 3 variables, N represents the number we need to find, `N3` is the cubed number and `N4` is the number to the fourth and we set a reasonable bound for each.
Afterwards, we simply need to call the previously defined function on `N3` and `N4` and appending all the resulting digits: we want that this new list contains all the digits from 0 to 9.
You can see the implementation of the problem in the source codes section A.4.

As we said in the introduction we have two possibilities, with the code shown at A.4 we achieve *"each digit must appear once"* constraint, but if we remove the `all_distinct` predicate we can achieve the *"each digit must appear* **at least** *once"* constraint.

A Source codes

A.1 Nurikabe - MiniZinc

```
% PARAMETERS
int: length;
array[1..length, 1..length] of 0..9: init;

% USER DEFINED PARAMETERS
int: water_cells = length * length - all_island_cells();
int: water_limit = water_cells div 2;

% VARIABLES
array[1..length,1..length] of var 0..max(init): solution;

%%% USEFUL PREDICATES AND FUNCTIONS
% Checking if a point is inside the grid of size length X length
predicate in_bounds(int: i, int: j) =
  i <= length /\ i >= 1 /\ j <= length /\ j >= 1;

% Checking if two points are k cells far from each other
predicate k_adjacency(int: distance_k, int:i, int:j, int:di, int:
   dj) =
  abs(di+dj) == distance_k /\ (abs(di) == 0 \/ abs(dj) == 0) /\
    in_bounds(i+di, j+dj);

% Encoding a tuple of coordinates to a single number (the opposite
   holds)
function var int: encode_point(var int: i, var int: j) =
  (i - 1) * length + (j-1);

% Checking the surface (island or water) is continuous
predicate surface_continuous(int: i, int: j, var set of int:
   visited, int: u, int: v, int: res) =
  res > 0 /\ (
    (i == u /\ j == v) \/
    exists (di,dj in -1..1 where k_adjacency(1,i,j,di,dj) /\ not (
       encode_point(i+di, j+dj) in visited)
          /\ solution[u,v] == solution[i+di,j+dj]) (
      (i+di == u /\ j+dj == v) \/ surface_continuous(i+di, j+dj, {
        encode_point(i,j)} union visited, u, v, res-1)
    )
  );

% The total number of islands cells
function int: all_island_cells() =
  sum (i,j in 1..length where init[i,j] > 0) (init[i,j]);

function var int: count_cells(int: val) =
  sum(k in 1..length) (count(solution[k,..], val));

predicate init_unique(int: val) =
  sum(k in 1..length) (count(init[k,..], val)) == 1;
```

```
%% ISLAND CONSTRAINTS
% Checking each cell of an island is surrounded by another cell of
    same island or by a water
predicate check_neighborhood(int: i,int: j) =
  forall (di,dj in -1..1 where k_adjacency(1,i,j,di,dj)) (
    solution[i+di,j+dj] == solution[i,j] \/ solution[i+di,j+dj] ==
        0
  );
predicate island_isolated() =
  forall(i,j in 1..length where solution[i,j] > 1)
    (check_neighborhood(i,j));


% The number of cells for each island is equal to the clue of that
    island
predicate island_count_cells() =
  forall(i,j in 1..length where init[i,j] > 0)(
    count_cells(init[i,j]) == init[i,j]
  );

% Each island must have a continuous shape
predicate island_continuous() =
  forall(i,j in 1..length where init[i,j] > 1) (
    forall(u,v in 1..length where init[i,j] == solution[u,v] /\ (i
        != u \/ j != v)) (
      surface_continuous(i,j,{}, u,v, init[i,j]-1)
    )
  );


%%% WATER CONSTRAINTS
% Adding water between two cells of different island which are
   diagonally adjacent
predicate diagonally_adjacents() =
  forall(i,j in 1..length where solution[i,j] > 0) (
    forall(di,dj in -1..1 where abs(di) + abs(dj) == 2 /\
       in_bounds(i+di,j+dj) /\
            solution[i,j] != solution[i+di, j+dj] /\ solution[i+di
               , j+dj] > 0) (
        solution[i+di, j] == 0 /\ solution[i, j+dj] == 0
    )
  );

% Adding water between two cells of different island which are 2
   cells far from the other horizontally
predicate horizontally_adjacents() =
  forall(i,j in 1..length where solution[i,j] > 0) (
    forall(dj in -2..2 where k_adjacency(2, i,j,0,dj) /\ solution[
       i,j] != solution[i, j+dj] /\ solution[i, j+dj] > 0) (
        solution[i, j+(dj div 2)] == 0
    )
  );

% Adding water between two cells of different island which are 2
   cells far from the other vertically
predicate vertically_adjacents() =
  forall(i,j in 1..length where solution[i,j] > 0) (
    forall(di in -2..2 where k_adjacency(2, i,j,di,0) /\ solution[
       i,j] != solution[i+di, j] /\ solution[i+di, j] > 0) (
        solution[i+(di div 2), j] == 0
    )
  );

% Water cannot create 2x2 cells shape
predicate water_not_squares() =
  forall(i,j in 1..length-1)(
    solution[i,j] + solution[i+1,j] + solution[i+1,j+1] + solution
       [i,j+1] > 0
  );

% The number of water cells is the difference between all the
   cells and islands' ones
predicate water_count_cells() =
  count_cells(0) == water_cells;

% Water must have a continuous shape
predicate water_continuous() =
  exists(i,j in 1..length where solution[i,j] == 0) (
    forall(u,v in 1..length where solution[u,v] == 0 /\ (i != u \/
        j != v)) (
      surface_continuous(i,j,{}, u,v, water_limit)
    )
  );

%%% ASSERTION
constraint forall(i,j in 1..length where init[i,j]>0) (assert(
   init_unique(init[i,j]), "init␣is␣not␣valid"));

%%% CONSTRAINTS
constraint forall(i,j in 1..length where init[i,j]>0) (solution[i,
   j]==init[i,j]);

constraint island_isolated();
constraint water_continuous();

constraint water_count_cells();
constraint island_count_cells();

constraint island_continuous();
constraint water_not_squares();

constraint diagonally_adjacents();
constraint horizontally_adjacents();
constraint vertically_adjacents();
```

```
solve :: int_search(solution, first_fail, indomain_min) satisfy;
% solve satisfy;

output [ show_float(4,0,solution[i,j]) ++
         if j == length then "\n\n" else "␣" endif |
         i in 1..length, j in 1..length
         ];
```

A.2 Ex2 - MiniZinc

```
include "globals.mzn";

int: n;
int: num_edges;
int: start;
int: target;

array[1..n] of var 0..n: x;
array[1..num_edges, 1..2] of 1..n: graph;

% It is the length of the path we are considering
var int: path_length = sum(i in 1..n) (x[i] > 0);

% y is the last element which is > 0. All elements after y is 0
predicate last_not_0(array[int] of var int: x, var int: y) =
  exists(i in 1..length(x)) (
    x[i] = y /\
    forall(j in i+1..n) (x[j] = 0) /\ % all elements after y are 0
    forall(j in 1..i) (x[j] > 0) % all elements before y are > 0
  );

% Generate all possible paths
predicate all_paths(array[int] of var int: x, array[int, 1..2] of
   var int: graph) =
  forall(i in 2..length(x)) (
    x[i] = 0 \/
    (
      x[i] > 0 /\ x[i-1] > 0 /\
      % Return index set of first dimension of two-dimensional
         array graph
      exists(j in 1..num\_edges) (
        (graph[j,1] = x[i-1] /\ graph[j,2] = x[i])
        \/
        (graph[j,1] = x[i] /\ graph[j,2] = x[i-1])
      )
    )
  );

% We can add path_length = n to see if we have a solution with
   that length
predicate compute_solution() =
  alldifferent_except_0(x) /\ x[1] = start /\
    last_not_0(x, target) /\ all_paths(x, graph);

constraint compute_solution();

%solve :: int_search(x, first_fail, indomain, complete) maximize
   path_length;
solve maximize path_length;

output [
  "Path␣length:␣\(path_length)␣-␣Path␣cost:␣\((path_length-1)*10)\
     n␣x:␣\([x[i]␣|␣i␣in␣1..n␣where␣fix(x[i])␣>␣0])\n"
];
```

A.3 Ex17 - MiniZinc

```
set of int: match = 0..9;
int: digit_limit = card(match);

var int: N;
var int: N1 = pow(N, 3);
var int: N2 = pow(N, 4);

constraint N > 0;
% Uncomment to relax the problem
% constraint int2float(N) < sqrt(sqrt(2147483647)); %2^31

function var int: digits(var int: num) =
  let {
    var 0..digit_limit: power;
    constraint num >= pow(10, power-1) /\ num < pow(10, power);
  } in power;

% Comment to relax the problem
% N1 and N2 must be lower than the cardinality of the set MATCH,
% because the concatenation of N1 and N2 is a permutation of the
   set MATCH.
constraint digit_limit == digits(N1) + digits(N2);

predicate check_occurence(int: i, var int: num) =
  exists (j in 1..digit_limit)
    (i == (num div pow(10, j-1) mod 10) /\ (num div pow(10, j-1) >
        0));

% If we want to relax the constraint we can use the \/
constraint forall (i in match)
  (check_occurence(i, N1) xor check_occurence(i, N2));

solve minimize N;
output [ show(N1) ++ "␣" ++ show(N2) ++ "␣" ++ show(N)] ;
```

A.4 Ex17 - Prolog

```
:- use_module(library(clpfd)).

numToList(NUM,[LIST|[]]):-
  NUM < 10,
  LIST is NUM,
  !.
numToList(NUM,LIST):-
  P is NUM // 10,
  numToList(P,LIST1),
  END is (NUM mod 10),
  append(LIST1,[END] ,LIST).

bocconi_17(VARS, ALL_DIG):-
  length(VARS, 3),
  VARS = [N, N3, N4],
  N in 1..200,
  N3 in 1..8000000,
  N4 in 1..1600000000,
  N3 #= N^3,
  N4 #= N^4,

  labeling([], VARS),
  numToList(N3, D3),
  numToList(N4, D4),
  append(D3, D4, ALL_DIG),
  all_distinct(ALL_DIG),
  subset([0,1,2,3,4,5,6,7,8,9], ALL_DIG).
```