CONSTRAINT AND ANSWER SET PROGRAMMING

FRANCESCO PINZAUTI

Abstract

The aim of this project is to solve a *Constraint Satisfaction Problem* using both *Answer Set Programming* and *MiniZinc*. On top of that a Python program was developed which allows the user to automatically choose to solve the problem with one of the two models. The user can also choose if the input should be generated randomly or inserted manually. In the first case he has control over some parameters of the problem (e.g. dimension of the grid).

CONTENTS

1 THE PROBLEM

The task is to construct non-crossing and non-overlapping paths that start from given numbered grid cells and meet each other in a single cell. The paths can make horizontal and vertical transitions between neighboring cells but must not pass numbered grid cells (except for their origins). Most importantly, for each path, the number in its origin cell prescribes the number of turns, that is, alternation between horizontal and vertical transition or vice versa, the path has to make.

If you need further details and an example, you can see the complete assignment explanation here.

2 PROJECT STRUCTURE

You can find the source code here and the source for the report here. The file structure of the project is the following:

In this section we try to give a quick look at the whole project, please note that if further details are needed all the code, both the models and the Python program, is properly documented.

```
src
├── asp ................................ ASP part of the project.
│   ├── data .....................................Input data.
│   │   ├── input.lp ......... User defined input for manual mode.
│   │   ├── benchmark1.lp .................. Benchmark instance.
│   │   ├── ...
│   │   └── benchmark30.lp ................. Benchmark instance.
│   └── model.lp ................................. ASP model.
├── minizinc ...................... MiniZinc part of the project.
│   ├── data .....................................Input data.
│   │   ├── input.dzn ........ User defined input for manual mode.
│   │   ├── benchmark1.dzn ................. Benchmark instance.
│   │   ├── ...
│   │   └── benchmark30.dzn ................ Benchmark instance.
│   ├── model.mzn ........................... MiniZinc model.
│   └── solver.mpc ............... MiniZinc solver configuration.
├── main.py. ............................. Program entrypoint.
├── asp.py. .................. Handles everything related to ASP.
├── mz.py. ............... Handles everything related to MiniZinc.
├── Makefile .. Define commands to setup and execute the project.
└── requirements.txt ...... Required packages to run the project.
```

2.1 Tech stack

Everything was developed, tested and executed on a local machine[1]. The project is built using Python 3.10.2, which allows to manage the ASP and MiniZinc models, the random instances and their solutions in only one command line interface.

Clingo 5.5.2 and a Visual Studio Code extension was used for ASP. All the project is built upon Clingo API [Pot22b], which was necessary

1 OEM: Xiaomi, OS: Windows 11 21H2, CPU: Intel Core i7-11370H, RAM: 16GB DDR4, GPU: NVIDIA GeForce MX450.

in order to generate and solve random instances and to format the output in an human readable way.

As far as MiniZinc goes the model was developed using MiniZinc 2.6.3 and MiniZinc IDE. Again, MiniZinc API [Min22a] was heavily used in the main script.

In both cases Github Copilot assisted the development process. This report is written using the Minted package for the source code and Inkscape for creating the images.

2.2 How to run

First thing first you need to create a virtual environment and install the required packages[2]. To do so just run the following commands:

The packages installed are, peraphs unsuprisingly, clingo and minizinc[dzn].

```
cd src/
make setup
```

If you want to know what you can do just type:

```
make help
```

We will however explain everything here.

2.2.1 *ASP*

Two modes are available: a random mode and a manual mode. The random mode will generate a random instance and solve it. The manual mode will allow you to define your own instance and solve it.

random mode If you want to use random mode you need to run the following command:

```
make asp-random  [d DIMENSION] [n NUMBER] [s SOLUTIONS]
```

you can choose the dimension of the board and the number of initial points. However if you don't want to the program will generate them randomly for you; you can also decide to input only the dimension or only the number of starting points. Note that a upper bound was set on the dimension, both if you manually define it or if you don't. Going beyond that could lead to excessive execution time[3]. By default all solutions will be displayed, if you want to display only a specific number of solution specify it to the argument `s`.

All of this is managed by the `asp_random()` function in the main.py file. It goes without saying that not all the randomly generated instances will be solvable, and the program will communicate when those are not.

2 It will detect automatically if you are using Windows or Linux, no need to adjust anything.

3 Why have you not used the –time-limit option? The API does not seem to support it. However all the benchmarks in section 6 respect the 5 minutes time limit as required.

manual mode If you want to use manual mode you need to run the following command:

```
make asp-manual [s SOLUTIONS]
```

this will solve the problem with the input data that you should enter in the file `src/asp/data/input.lp`. The `s` argument is the same as explained above. All of this is managed by the `asp_manual()` function in the main.py file. Some sanity checks on the user input are performed, but it should still be reasonable in order to produce something meaningful and in an acceptable time.

As anticipated the solutions will be printed in an human readable way thanks to the function `asp_prettier()` located in the main.py file.

2.2.2 *MiniZinc*

Again, two modes are available: a random mode and a manual mode. The parameters and the functioning are the same, however we briefly present them.

If the program fails try adding MiniZinc installation location to your PATH. More on that here.

random mode If you want to use random mode you need to run the following command:

```
make minizinc-random  [d DIMENSION] [n NUMBER] [s SOLUTIONS]
```

where `d` is the dimension of the board, `n` is the number of starting points and `s` is the number of solutions to be displayed. All the parameters are optional, if not inserted `d` and `n` will be randomly chosen by the program, and all solution will be displayed.

manual mode If you want to use manual mode you should insert the input data in the file `src/minizinc/data/input.dzn` and run the following command:

```
make minizinc-manual [s SOLUTIONS]
```

where `s` is the number of solutions to be displayed.

2.2.3 *Both*

You can also solve both the ASP and the MiniZinc problems at the same time, inserting the input data in the file `src/asp/data/input.lp` and `src/minizinc/data/input.dzn`. The command is

```
make both [s SOLUTIONS]
```

where `s` as always is the number of solutions to be displayed.

3 ASP MODEL

As explained in [Pot22a] an ASP model is divided in four part: generate, define, test and display. As for the generate part our Python program handles both the generation of the random instances in random mode and the inclusion of the user defined input in manual mode.

3.1 Generate

The input is composed by `n` + 1 *ground terms*, with `n` the number of starting points:

```
boardl(D).
number(X_1, Y_1, T_1).
.
.
.
number(X_n, Y_n, T_n).
```

where `D` is the dimension of the board and `X_i, Y_i` are the coordinates[4] of the starting points. `T` is the number of turns each starting point has to make in order to arrive to the intersection point.

3.2 Define

The define part is displayed below

TLDR: There can be only one intersection point (i.e. the goal) and every two points in the grid there can or can't be a link between them. We then define with denials when those links can exist.

```
1  grid(1..D, 1..D) :-boardl(D).
2  1 {goal(X,Y): grid(X,Y)} 1.
3
4  {link(start(X1, Y1), end(X2, Y2)) :  grid(X1,Y1),
   ↪  grid(X2,Y2), |X1 - X2| + |Y1 - Y2| = 1}.
5
6  connected(start(X1,Y1) ,end(X2,Y2)) :- link(start(X1,Y1),
   ↪  end(X2,Y2)).
7  connected(start(X1,Y1), end(X2,Y2)) :-
   ↪  connected(start(X1,Y1), end(Z1,Z2)),
   ↪  link(start(Z1,Z2),end(X2,Y2)).
```

In line 1 we have a rule for defining the *predicate* grid, i.e. a matrix of dimension DxD. The goal point, i.e. the intersection point of our links is defined in line 2: this has to be one and has of course to be part of the grid. On line 4 we have a rule for defining the predicate `link`, which is the connection between two points of the grid, this connection

4 From now on we represent column position with X and row position with Y. Even in the Python script, when a point is labelled as (X, Y) we are referring to columns and row. This is rather unusual, but necessary in order to be consistent with the notation of the problem assigned.

can take place or not and can't be diagonal but only horizontal and vertical. Note the use of the predicates `start` and `end` which serves the scope to give directionality to the links. On line 6 we have the `connected` predicate, defined recursively on the definition of `link`, and it is the predicate that checks if two points are connected.

3.3 Test

Let us start describing the first *denials*:

```
1  :- #count{grid(X,Y): number(X,Y, _)} < 2.
2
3  :- goal(X, Y), number(X, Y, _).
4
5  :- goal(X, Y), not connected(start(X0, Y0), end(X,Y)),
   ↪  number(X0,Y0, _).
```

In line 1 we impose the number of starting point to be at least 2. In line 3 we impose that the goal point must not be a starting point. In line 5 we impose that the goal point must be connected to a starting point.

```
1  :- link(_, end(X0, Y0)), number(X0, Y0, _).
2
3  :- link(start(X,Y), _), not link(_, end(X,Y)), not
   ↪  number(X,Y, _).
4
5  :- link(_, end(X,Y)), not link(start(X,Y), _), not goal(X,
   ↪  Y).
```

In line 1 we impose that no link can end in a starting point. In line 3 we impose that a link can start in a point only if in that point another link ends or if it is the starting point. In line 5 we impose that a link can end in a point only when in that point another link starts or if it is the goal.

Please note that some of the denials here are partially redundant, however as stated in [Pot22a] and as verified experimentally often adding more denials, even if overlapping, results in a better performance.

```
1  :- link(start(X, Y),end(X1, Y1)), link(start(X, Y),
   ↪  end(X2, Y2)), X1 != X2.
2
3  :- link(start(X, Y),end(X1, Y1)), link(start(X, Y),
   ↪  end(X2, Y2)), Y1 != Y2.
4
5  :- link(start(X1, Y1),end(X, Y)), link(start(X2, Y2),
   ↪  end(X, Y)), X1 != X2, not goal(X,Y).
6
7  :- link(start(X1, Y1),end(X, Y)), link(start(X2, Y2),
   ↪  end(X, Y)), Y1 != Y2, not goal(X,Y).
```

In line 1 and 3 we impose that there can not be two links from one starting point. In line 5 and 7 we impose that two links can't end in the same point (i.e. there can not be intersections) if the point is not the goal.

```
1  :- link(start(X1, Y1), end(X2, Y2)), link(start(X2,Y2), end(X1, Y1)).
2
3  :- link(_, end(X1,Y1)), not connected(start(X1,Y1), end(X, Y)), goal(X, Y), not goal(X1,Y1).
```

In line 1 we impose that a link can not end when it starts. In line 3 we impose that a link have to be connected to the goal. All together, we are avoiding loops.

```
1  :- #count{grid(X2,Y2), grid(X3, Y3): link(start(X1, Y1), end(X2, Y2)), link(start(X2, Y2), end(X3, Y3)), |X1-X3| = 1, |Y1-Y3| = 1, connected(start(X0, Y0), end(X1, Y1));
2  grid(X1,Y1), grid(X2, Y2): link(start(X0, Y0), end(X1, Y1)), link(start(X1, Y1), end(X2, Y2)), |X0-X2| = 1, |Y0-Y2| = 1} != T, number(X0, Y0, T).
```

Finally we impose, using the *aggregate* `#count`, that for each path from each starting point to the goal the number of turns should be the one prescribed. We do that by checking two consecutive links connecting three points, if the difference between the x-axis and y-axis coordinates of the first and third cell is both 1, then we have a turn.

Note that we make use of *pooling* in order to consider two cases: the first case is when the turn is made by a a triple of cells where none of them is a starting point (but they must be obviously connected to it), the second case is when the turn is made by a triple directly containing a starting point.

3.4 Display

Finally with the last two lines we display our solution, hiding everything except the goal and the link towards it. The output format is:

```
1  link(X_1, Y_1, X_2, Y_2).
2  .
3  .
4  .
5  link(X_{n-1}, Y_{n-1}, X_n, Y_n).
```

where link is the predicate defined in 3.2 which describes the links between points. All the links together describe the path from the starting points to the intersection point.

4 MINIZINC MODEL

The approach taken with MiniZinc is completely different. Data structures used to represent the same data and relations are different, input and output format are completely different.

4.1 Data structures

Let us introduce everything with order:

```
1 par int: dimension; set of int: D = 1..dimension;
2 par int: number; set of int: N = 1..number;
3 array[N, 1..3] of par 0..dimension: starting_points;
4 array[D, D, 1..2] of var 0..12: Board;
```

On line 1 we define the dimension of the board as a parameter named dimension, and a set of integers D on top of that. Then, on line 2, we define also the number of starting points[5] and again a set of integers N on top of that. On line 3 we define the matrix starting_points, which has the following input format:

```
starting_points = [|X₁, Y₁, T₁
                       ...
                    |Xₙ, Yₙ, Tₙ|];
```

with n the number of starting points and each line representing, in order, the X, Y coordinates of a starting point and the number of turns it has to make in order to arrive to the goal.

Finally, on line 4, we define the multidimensional array of variables Board. Board is of size DxD and has two dimensions: the first one will store in each position one of the possible elements among the one represented in figure 1.

The second dimension of the board will keep track of which path[6] each cell belongs to, this is necessary, as we will se later, in order to constraint the number of turns each path has to make.

Below you see the output of Board for the assignment example[7]:

Board[i,j,1]:

0	0	12
4	5	9
12	12	1

5 This is something not necessary in ASP, but it is necessary in MiniZinc as we are using a multidimensional array to represent the starting points and MiniZinc does not support dynamically sized arrays.

6 We have one path for each starting point, all paths intersect in one single point at the end.

7 The second dimension of the Board is actually hidden in the output as not really relevant and only useful for internal checks. However if you want to show it it is sufficient to replace `show(Board[i,j,1]) | i,j in D` with `show(Board[i,j,t]) | t in 1..2, i,j in D`.

`Board[i,j,2]`:

0	0	3
1	1	4
1	2	2

Note that the cell labelled, in the second dimension of the board, as `number + 1` (in this example 4) is the cell containing the intersection of all the paths.

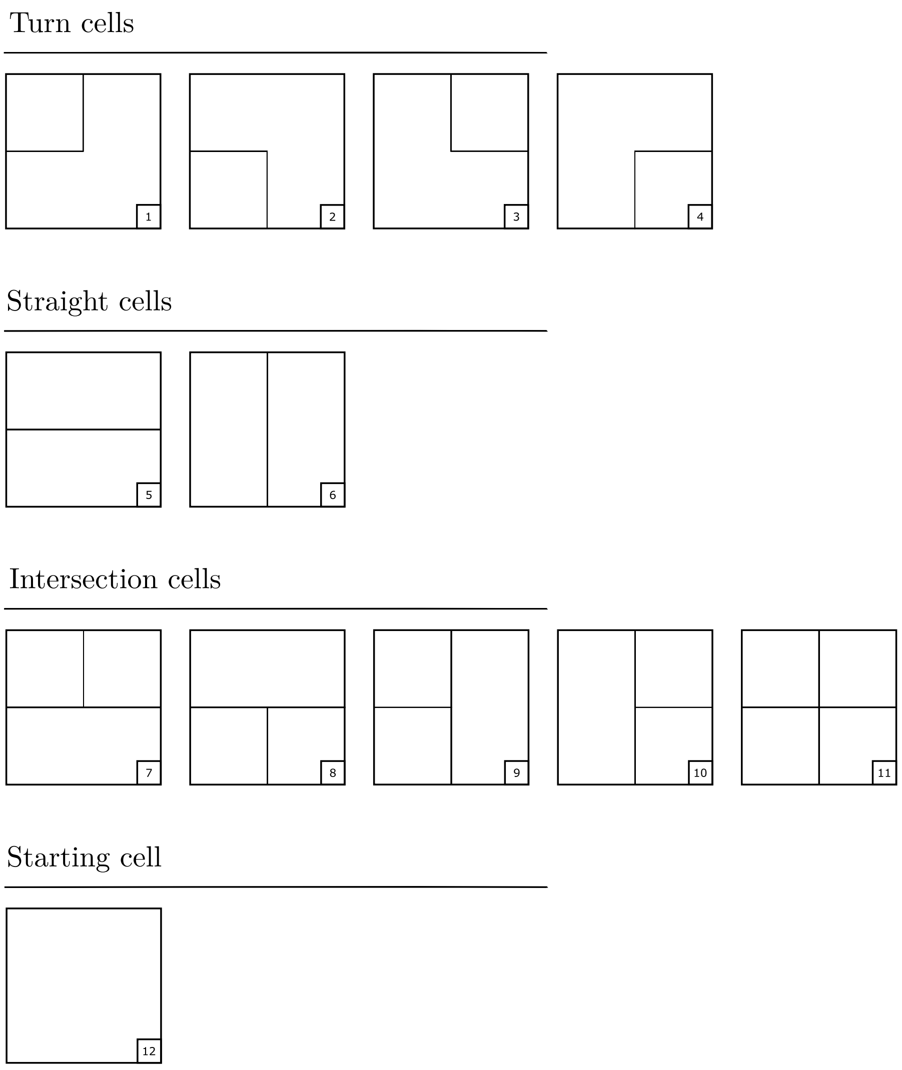

Figure 1: First dimension of Board. Each number corresponds to the element represented.

4.2 Predicates

We start defining the predicates `t_cells(i,j)`, `b_cells(i,j)`, `l_cells(i,j)` and `r_cells(i,j)`. These predicates defines the possible cells, again among the ones in figure 1, that we can find on the top, on the bottom, on the left and on the right of another cell. Let us take an example of one of this predicate:

```
predicate t_cells(int: i, int: j) =
(Board[i-1, j, 1] = 2 \/ Board[i-1, j, 1] = 4 \/
↪  Board[i-1, j, 1] = 6 \/ Board[i-1, j, 1] = 8 \/
↪  Board[i-1, j, 1] = 9 \/ Board[i-1, j, 1] = 10 \/
↪  Board[i-1, j, 1] = 11);
```

On top of that we build the predicates `top(i,j)`, `bottom(i,j)`, `left(i,j)` and `right(i,j)`. Let us again take an example of one of those predicates:

```
predicate top(int: i, int: j) =
i-1 > 0 /\
(t_cells(i,j)\/ Board[i-1, j, 1] = 12) /\
(Board[i-1, j, 2] != number + 1 -> (Board[i, j, 2] =
↪  Board[i-1, j, 2]  \/ Board[i, j, 2] = number + 1));
```

We are in position `i`, `j` and we want to state the possible elements on top of that cell. First of all we check to be still inside the board, i.e. that the position $i - 1$ is greater than zero. Then we proceed allowing all the cells that can be connected on top and this is done in `t_cells(i,j)`. This was for the first dimension of the board. As far as the second dimension we check that the connected cell is not an intersection cell, the intersection cell is labelled on the second board with the number `dimension + 1`. If it is not we require the cell (`i`, `j`) to be on the same path of the connected cell, which in this case is the cell (`i-1`, `j`), or the be an intersection cell. In this way we keep track of each path, and allow them to end in the same point.

4.3 Constraints

We shall start placing in each starting point a cell of type 12 (again, according to figure 1 which from now on will be taken as reference when stating any cell number). Note that in order to be as consistent as possible with the ASP model we have to adapt the peculiar notation of the assignment (i.e. inverse row index, column and row notation inverted) to the MiniZinc notation. The following code does so:

```
forall(n in N)(
if starting_points[n,2] = 1 then Board[dimension,
↪  starting_points[n,1], 1] = 12 /\ Board[dimension,
↪  starting_points[n,1], 2] = n
elseif
starting_points[n,2] = dimension then Board[1,
↪  starting_points[n,1], 1] = 12 /\ Board[1,
↪  starting_points[n,1], 2] = n
else
Board[dimension - starting_points[n,2] + 1,
↪  starting_points[n,1], 1] = 12 /\ Board[dimension -
↪  starting_points[n,2] + 1, starting_points[n,1], 2] = n
endif
)
```

In the first dimension of the board we place the number 12 while in the second dimension we label each starting point with a number, this number will define each path on the board.

We proceed now using the predicates of 4.2 in order to define which moves are allowed for each type of cell.

```
forall(i,j in D)(
    (Board[i, j, 1] = 1 -> left(i,j) /\ top(i,j)) /\

    ...

    (Board[i, j, 1] = 12 -> top(i,j) \/ bottom(i,j) \/
↪  right(i,j) \/ left(i,j))
)
```

We reported only two examples, the cell 1 allow to move on the left and on top, while the starting point cell allows to move everywhere. All the other cells allowed moves are properly defined.

Another constraint that we impose (and the code of which we omit, as it is quite trivial) is that if a cell is empty in the first dimension of the board it should be the same even in the second dimension and viceversa.

In addition we impose that on the second dimension of the board there can't be numbers (i.e. paths) greater than `number + 1`.

The last constraint avoid multiple paths from a single starting point, we partially show it below:

```
forall(i,j in D where Board[i,j,1] = 12) (
if top(i,j) then if i+1 <= dimension then not b_cells(i,j)
↪  endif /\ if j-1 > 0 then not l_cells(i,j) endif /\ if
↪  j+1 <= dimension then not r_cells(i,j) endif

...

elseif right(i,j) then if i-1 > 0 then not t_cells(i,j)
↪  endif /\ if i+1 <= dimension then not b_cells(i,j)
↪  endif /\ if j-1 > 0 then not l_cells(i,j) endif
else true endif
)
```

4.4 Global constraints

The following global constraints were used:

among: `predicate among(var int: n, array [$X] of var int: x, set of int: v)` Requires exactly `n` variables in `x` to take one of the values in `v`.

count_eq: `predicate count_eq(array [$X] of var int: x, var int: y, var int: c)` Constrains `c` to be the number of occurrences of `y` in `x`.

The use of let expression was also required, we report the syntax below:

```
<let-expr> ::= "let" "{" <let-item> ";" ... "}" "in"
↪  <expr>

<let-item> ::= <var-decl-item>
             | <constraint-item>
```

Let us explain how global constraints were used starting with:

```
let {array[D,D] of var 0..12: MoveBoard; constraint
↪  forall(i,j in D) (MoveBoard[i,j] = Board[i,j, 1]);} in
if number = 2 then
among(0, MoveBoard, 7..11)
elseif number = 3 then
among(1, MoveBoard, 7..10) /\ count_eq(MoveBoard, 11, 0)
elseif number = 4 then
count_eq(MoveBoard, 11, 1) /\ among(0, MoveBoard, 7..10)
else
false
endif
```

Here we require that, depending on the number of starting points, there should be a certain type of intersection cells, e.g. if we have four starting points we need a single interception cell, the number 11. And so on and so forth for the other cases.

We proceed now with three brief constraints:

Some of those constraints could seem redundant, but for how the predicates in section 4.2 were defined they turned to be necessary.

```
forall (n in N)(
let {array[D,D] of var 0..12: MoveBoard; constraint
↪  forall(i,j in D) (if  Board[i,j,2] = n then
↪  MoveBoard[i,j] = Board[i,j, 1] else MoveBoard[i,j] = 0
↪  endif);} in
among(starting_points[n,3], MoveBoard, 1..4)
)
```

Here we require each path to do the prescribed number of turns: the number of turns for each starting point is stored in the third column of starting_points and the moves from 1 to 4 are, according to figure 1, the turning cells.

```
let {array[D,D] of var 0..12: PathsBoard; constraint
 ↪  forall(i,j in D) (PathsBoard[i,j] = Board[i,j,2]);} in
count_eq(PathsBoard, number + 1, 1)
```

Here we impose that there has to be one and only one intersection point (i.e. point labelled as `number + 1`).

```
let {array[D,D] of var 0..12: MoveBoard; constraint
 ↪  forall(i,j in D) (MoveBoard[i,j] = Board[i,j,1]);} in
count_eq(MoveBoard, 12, number)
```

Then with the last constraint we impose the number of starting cells (i.e. cells labelled with 12 in the first dimension of the board) to be exactly the same as the number of starting points.

5 CONCLUSIONS

The following configuration was used for MiniZinc:

MiniZinc solver configuration can be found in `src/minizinc/solver.mpc`.

solver: Chuffed 0.10.4

number of solutions: All solutions

optimization level: -O0 (no optimization)

time limit: 300 s

annotations: `int_search(Board, first_fail, indomain_median) satisfy;`

This set of choices was the result of a series of experiments on both small and large instances. As a result this was the combination of parameters that seems to perform better for this specific problem.

As we will see in section 6 a set of 30 istances were created and solved. Keep in mind that not all results were manually checked for their correctness, only the ones with dimension of the Board equal to 3; however the equal number of solutions between ASP and MiniZinc of each instance was taken as a good indicator of the correctness of the models.

The order of magnitude of execution times is way below the one required, this is intended, as it was considered meaningless to go above a reasonable dimension of the board.

ASP is, as expected, considerably faster than MiniZinc, but not in all cases: when the number of solutions increases the difference between the two solvers decreases until MiniZinc seems to actually outperform ASP.

It must be however noted that given the almost negligible deltas between execution times, most of the differences between the instances can be attributed to statistical fluctuation and/or different performance of the system at different times.

6 BENCHMARKS

You can find all the benchmark instances in the `src/minizinc/data` folder and `src/asp/data` folder. We briefly report an overview of the performance on the table below.

All the instances were randomly generated by the Python script.

Benchmark instances					
#	Dimension	Starting points	Minizinc Time	ASP Time	Solutions
1	3	2	506 ms	5 ms	4
2	3	2	516 ms	16 ms	3
3	3	3	367 ms	4 ms	1
4	3	3	350 ms	8 ms	1
5	3	3	369 ms	7 ms	1
6	4	2	363 ms	13 ms	3
7	4	2	567 ms	105 ms	41
8	4	2	566 ms	121 ms	35
9	4	3	652 ms	20 ms	1
10	4	3	626 ms	19 ms	4
11	4	3	650 ms	20 ms	2
12	4	3	599 ms	22 ms	3
13	4	4	585 ms	27 ms	1
14	4	4	565 ms	15 ms	1
15	4	4	522 ms	13 ms	2
16	5	2	412 ms	185 ms	8
17	5	2	774 ms	852 ms	66
18	5	2	391 ms	89 ms	36
19	5	3	425 ms	66 ms	9
20	5	3	379 ms	31 ms	3
21	5	3	715 ms	35 ms	2
22	5	3	876 ms	96 ms	5
23	5	4	707 ms	50 ms	1
24	5	4	573 ms	41 ms	1
25	5	4	554 ms	60 ms	1
26	6	2	897 ms	3483 ms	80
27	6	3	810 ms	509 ms	15
28	6	3	702 ms	193 ms	3
29	6	4	840 ms	155 ms	2
30	6	4	673 ms	141 ms	1

REFERENCES

[Cor+22] Thomas H. Cormen et al. *Introduction to Algorithms, 4th Edition.* MIT Press, 2022.

[Dov16] Agostino Dovier. *Answer Set Programming (ASP) e la codifica dei rompicapi.* May 2016.

[Kje22] Håkan Kjellerstrand. *MiniZinc solved problems.* `http://www.hakank.org/minizinc/`. 2022.

[Min22a] MiniZinc. *MiniZinc Python — MiniZinc Python 0.6.1 documentation.* 2022.

[Min22b] MiniZinc. *The MiniZinc Handbook — The MiniZinc Handbook 2.5.5.* 2022.

[Pot22a] Potassco. *An introduction to our Answer Set Programming tools focusing on gringo, clingo, and clasp.* 2022.

[Pot22b] Potassco. *Clingo API documentation.* 2022.