Minizinc Tutorial

Search and Planning, MEIC, 2022/23

1 Introduction to MiniZinc

MiniZinc is a language for specifying constraint satisfaction and optimization problems over integers and real numbers. A MiniZinc model does not dictate how to solve the problem, it does however translate a problem description into different forms suitable for different solvers, such as Constraint Programming (CP), Mixed Integer Programming (MIP) or Boolean Satisfiability (SAT) solvers.

MiniZinc models are written in a way similar to mathematical formulations – using sums over index sets, logical connections, if-then-else statements, etc – that are fed to a supported solver to obtain a solution for the intended problem.

A model in MiniZinc is composed by four main components:

1. variables;
2. constraints;
3. solve statement;
4. output statement.

1.1 Variables

In MiniZinc there are two types of variables: *parameter* variables and *decision* variables. As MiniZinc is a typed language, all variables must be explicitly given a type when declared; in MiniZinc the basic parameter types are integers (int), floating point numbers (float), booleans (bool) and strings (string).

Parameter variables (or constant variables) are variables that must be assigned a value manually and can only be assigned a value once. They can be declared as follows:

```
<var-type> : <var-name>;
```

if the type being used is either int or float the variable can also be declared as:

```
<l> .. <u> : <var-name>;
```

where <*l*> and <*u*> are, respectively, the fixed lower and upper bounds of the variable. As an example, let's imagine we want to define a constant for the number of students in a class; this could be done as:

```
int : n_students;
n_students = 20;
```

or

```
int : n_students = 20;
```

in this example we declare the variable *n_students* as being an integer and assign it the value of 20.

Decision variables are variable with an unknown value, which the solver will assign when executed, assigning a value (if possible) that satisfies the constraints of the model. Decision variables can be declared as follows:

```
var <var-type> : <var-name>;
```

if the type being used is either int or float the variable can be also be declared as:

```
var <l> .. <u> : <var-name>;
```

where again <*l*> and <*u*> are fixed lower and upper bounds of the variable. As an example, let's imagine we want to define a variable for the number of students approved in class, this could be done as:

```
var int : students_approved;
```

in this example we declare the variable *students_approved* that will be assigned a value when we give the model to a solver.

MiniZinc also supports more complex variable types such as sets and arrays.

1.2 Constraints

In MiniZinc, the constraints of a model define the expressions that the decision variables must satisfy to find a solution for the problem. These expressions are Boolean and as such must be evaluated to either *True* or *False*. A constraint expression is declared as:

```
constraint <expression>
```

As an example take a constraint that states that a student to pass class must have an average grade higher or equal than 9.5:

```
constraint average_grade >= 9.5;
```

The expressions in a constraint can be as simple as an equality test, as in the example above, using one of the relational operators provided in the language – equal (==), not equal (!=), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=) – or some more complex expressions that involve *for-loops*, *if-then-else* statements, among others.

1.3 Solve statement

The solve statement defines the type of problem the model has to solve to obtain a solution. MiniZinc supports three types of solutions to solve: satisfaction problems (*satisfy*), maximization problems (*maximize*) and minimization problems (*minimize*).

A satisfaction problem statement is declared as follows:

```
solve satisfy;
```

For maximization and minimization problems the statement is declared as

```
solve maximize <arithmetic expression>;
solve minimize <arithmetic expression>;
```

1.4 Output statement

The final component of a model in MiniZinc is the output statement. This statement defines how the model outputs the final solution and is composed by a sequence of string separated by commas. A typical output statement is declared as follows:

```
output [<string expression>, ..., <string expression>];
```

1.5 Additional MiniZinc Information

The information in this section aims at given the foundations for understanding how to program in the MiniZinc language.

For more detailed information we advise to see the following websites:

- MiniZinc Documentation - `https://www.minizinc.org/doc-2.5.5/en/part_4_reference.html`
- MiniZinc Tutorials - `https://www.minizinc.org/doc-2.5.5/en/part_2_tutorial.html`

2 Exercise 1 - Color Coding Australia

For this exercise, take the picture of Australia in Figure 1 and consider we want to color each of its seven states in such a way that no state has the same color as an adjacent state. However we are limited to using only three colors, is there a solution that satisfies the coloring restrictions of the problem?

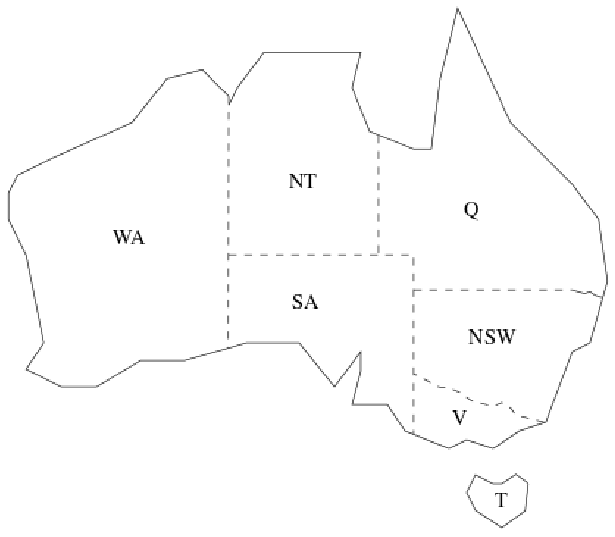

Figure 1: Map of Australia

Hints:

- coding the colors as integers simplifies the problem.
- problem can be solved with 9 constraints.

3 Exercise 2 - Baking Cakes

The first exercise focused on understanding the basic concepts of MiniZinc of how to declare different types of variables, identify which variables are needed to solve a problem and how to build constraint expressions with relational operators.

However not all problems are solved by satisfying simple inequality relational operations. In this exercise we will focus in arithmetic operations and in finding solutions that better optimize arithmetic problems.

For this exercise you are hired to bake chocolate and banana cakes for a party. A banana cake takes 250g of flour, 2 mashed bananas, 75g sugar and 100g of butter. A chocolate cake takes 200g of flour, 75g of cocoa, 150g sugar and 150g of butter.

In your pantry you currently have 4kg flour, 6 bananas, 2kg of sugar, 500g of butter and 500g of cocoa.

For each chocolate cake you make for the party you are paid 4.5€ and for each banana cake 4€. How many cakes of each must you bake in order to maximize your profit?

Hints:

- You want to maximize the number of cakes.
- Constraints can be mathematical expressions like $x + y < z$.

4 Exercise 3 - Job Scheduling

In this exercise you will be responsible for scheduling the working times of different machines to complete a series of tasks in the least time possible.

In this exercise there are five tasks – T1, T2, ..., T5 – and each task takes one hour to complete. The tasks can be executed in time slots 1, 2 and 3, each corresponding to a different hour, and different tasks can be executed simultaneously, however there are some restrictions:

- T1 must start after T3;
- T3 must start before T4 and after T5;
- T2 cannot be executed simultaneously with T4 or T1;
- T4 cannot start in time slot 2.

5 Exercise 4 - Sudoku

In this exercise you will create a model that can solve any given Sudoku problem.

A Sudoku problem is composed by one overall square – main square –, sub-divided in smaller squares, with the same number of squares on along each of the edges of the main square. The aim of a Sudoku problem is to fill each sub-square with numbers from 1 to N, where N is the maximum number that can appear in the sub-square. However to have a correct solution the following restrictions must be true:

- in each sub-square there must be no repeated numbers.
- in each row a number must appear only once.
- there cannot be repeated numbers in each column.

Hints:

- the predicate *alldifferent* – you must import the predicate with either the command *include "globals.mzn"* or *include "alldifferent.mzn"* – can be used to guarantee there are no repeated numbers.
- the function *forall* can be used to iterate over arrays, creating aggregations of elements. *forall* can be used in the form

$$forall(< generator\_expression >)(< expression >)$$