MiniZinc Basic Guide

MiniZinc is a language for specifying combinatorial optimization problems. The first step in solving a combinatorial problem is to precisely define the problem. As part of the assignment, you are asked to model problems using MiniZinc.

The MiniZinc language lets users write models in a way that is close to a mathematical formulation of the problem, using familiar notation such as existential and universal quantifiers, sums over index sets, or logical connectives like implications and if-then-else statements. Furthermore, MiniZinc supports defining predicates and functions that let users structure their models (similar to procedures and functions in regular programming languages). The extension .mzn used to indicate a MiniZinc model files whereas .dzn is used for data files. Data files can only contain assignment statements for decision variables and parameters in the model(s) for which they are intended.

This document is meant to provide a quick start and introduction to MiniZinc and it is compiled by following the official MiniZinc Documentation[1]. Please refer to MiniZinc Documentation for a more detailed understanding of MiniZinc.

MiniZinc Basics:

1. **Parameters**

Parameters are similar to (constant) variables in most programming languages. They must be declared and given a type. They are given a value by an assignment. MiniZinc allows the assignment to be included as part of the declaration (as in the line above) or to be a separate assignment statement.

For example:

$$\text{int: a} = 2;$$

is equivalent to

$$\text{int: a};$$
$$\text{a} = 2;$$

Unlike variables in many programming languages a parameter can only be given a single value, in that sense they are named constants. The basic parameter types are integers (int), floating point numbers (float), Booleans (bool) and strings (string). Arrays and sets are also supported.

[1] https://docs.minizinc.dev/en/stable/

2. Decision Variables

MiniZinc models can also contain another kind of variable called a decision variable. These variable represent the decisions that typically define the solution of the problem.

Decision variables are variables in the sense of mathematical or logical variables. Unlike parameters and variables in a standard programming language, the modeller does not need to give them a value. Rather the value of a decision variable is unknown and it is only when the MiniZinc model is executed that the solving system determines if the decision variable can be assigned a value that satisfies the constraints in the model and if so what this is.

For each decision variable we need to give the set of possible values the variable can take. This is called the variable's domain. This can be given as part of the variable declaration and the type of the decision variable is inferred from the type of the values in the domain. Decision variables are declared with var prefix.

For example, an integer decision variable is declared as either:

$$\text{var int : var-name}$$
$$\text{var l..u : var-name}$$

where l and u are integers depicting the domain of the int variable.

Array and set data structures can also be used for defining parameters and decision variables. Sets and arrays can be defined and instantiated as below:

$$\text{set of int: <set name>} = 0..\text{h};$$

where the set comprises of integers from $0..\text{h}$.

$$\text{array}[1..\text{NUM ROWS},1..\text{NUM COLUMNS}] \text{ of var float: <array name>};$$

where $<\text{array name}>$ is a 2D array and $[1..\text{NUM ROWS},1..\text{NUM COLUMNS}]$ are its dimensions. The element of the array in the i^{th} row and j^{th} column can be accessed using an expression $\text{t}[\text{i},\text{j}]$.

3. Constraints

The next component of the model are the constraints. These specify the Boolean expressions that the decision variables must satisfy to be a valid solution to the model. Relational operators like equal (= or ==), not equal (!=), strictly less than (<), strictly greater than (>), less than or equal to (<=), and greater than or equal to (>=) can be used to define constraints. Constraint items form the heart of the model. They have the form:

$$\text{constraint } \langle \text{Boolean expression} \rangle;$$

For example, to write a constraint on an array A where

$$\sum_{j=1}^{t} \mathrm{A[j]} = 1$$

the constraint can be written as

$$\text{constraint sum(j in 1..t) (A[j])} = 1;$$

4. Solve

Solve items specify exactly what kind of solution is being looked for. They have one of three forms:

solve satisfy;
solve maximize <arithmetic expression>;

solve minimize <arithmetic expression>;

Solve satisfy is used when we wish to find a value for the decision variables that satisfies the constraints but we do not care which one. A model is required to have at most one solve item. If it's omitted, it is treated as solve satisfy.

5. Output

The final part of the model is the output statement. This tells MiniZinc what to print when the model has been run and a solution is found. An output statement is followed by a list of strings. These are typically either string literals which are written between double quotes and use a C like notation for special characters, or an expression of the form $\mathrm{show(e)}$ where e is a MiniZinc expression.

Additional Resource (Recommended!)

To learn more about how to do modeling in MiniZinc and prepare for the assignment, please refer to the following sections of the MiniZinc tutorial[2]:

2.1. Basic modelling in MiniZinc
- 2.1.1. Our First Example
- 2.1.2. An Arithmetic Optimisation Example
- 2.1.3. Datafiles and Assertions
- 2.1.4. Real Number Solving *(optional)*
- 2.1.5. Basic structure of a model

2.2. More Complex Models
- 2.2.1. Arrays and Sets
- 2.2.2. Global Constraints
- 2.2.3. Conditional Expressions
- 2.2.4. Enumerated Types *(optional)*
- 2.2.5. Complex Constraints *(optional)*
- 2.2.6. Array Access Constraints
- 2.2.8. Set Constraints *(optional)*
- 2.2.9. Putting it all together

2.3. Predicates and Functions
- 2.3.1. Global Constraints *(see all-different and cumulative, others are optional)*

2.5. Tuple and record types
- 2.5.1. Declaring and using tuples and records
- 2.5.2. Using type-inst synonyms
- 2.5.3. Types with both var and par members *(optional)*
- 2.5.4. Example: rectangle packing using records *(optional)*

2.7. Effective Modelling Practices in MiniZinc *(optional)*
- 2.7.1. Variable Bounds *(optional)*
- 2.7.2. Effective Generators *(optional)*
- 2.7.4. Modelling Choices *(optional)*
- 2.7.6. Symmetry *(optional)*

The sections marked as optional are not necessary for this course and are intended for advanced students interested in obtaining a better understanding of modelling.

[2] https://docs.minizinc.dev/en/stable/part_2_tutorial.html