Elementary MiniZinc for Mathematics and Puzzles

Math problems and puzzles solved with MiniZinc

Charles Joscelyne

October 15, 2020

Preface

MiniZinc *https://www.MiniZinc.org/* is a free and open source constraint solving programming language. Concisely put, MiniZinc allows a user to model a quantitative problem as a collection of constraints expressed in terms of modeler defined decision variables. Once the constraints have been defined, MiniZinc calls on its library of pre-programmed solvers to find one or more solutions of the variables satisfying the constraints. Being more akin to a functional programming language than an imperative one, MiniZinc transfers the burden of writing a specific combinatorial or optimization algorithm imperatively to that of expressing a given problem functionally in terms of MiniZinc language constructs.

By way of example let us consider problem 13 from the 2019 American Mathematics Competition 8 Examination. *https://artofproblemsolving.com/wiki/index.php/2019_AMC_8_Problems*

```
A palindrome is a number that has the same value when read from
left to right or from right to left. (For example, 12321 is a
palindrome.) Let N be the least three-digit integer which is not a
palindrome, but which is the sum of three distinct two-digit
palindromes. What is the sum of the digits of N?


%Decision variables
var 1..9:N1;
var 0..9:N2;
var 0..9:N3;

var 1..9:A;
var 1..9:B;
var 1..9:C;


%Constraints variables
constraint N1 != N3;
constraint A != B;
constraint A != C;
constraint B != C;
constraint 100*N1 + 10*N2 + N3 = 10*A+A+10*B+B+10*C+C;
```

```
%solve
solve minimize 100*N1 + 10*N2 + N3;

%Display solution
output [ "\(N1)"++" "++"\(N2)"++" "++"\(N3)"++" sum of digits = "+.
+"\(N1+N2+N3)"];
% 1 1 0 sum of digits = 2
```

The solution: 110 = 77 + 22 + 11 to the palindrome problem is referenced as a MiniZinc comment in the program's last line.

A cursory glance of the program above shows the three components of any elementary MiniZinc model:

1. Definition of the model's decision variable(s).
2. Definition of model's decision constraints.
3. A solve statement.

Although an output statement is included in above program, MiniZinc will usually display useful results in the absence of one. Nonetheless, a MiniZinc modeler most likely will want to control the contents and format of any output. MiniZinc's output statement enables a modeler to do so.

Before proceeding to the study of individual problems and their solution in MiniZinc, I should note that this tract is not entirely intended to be a tutorial on MiniZinc. Although, I expect the reader would be able to learn various features of the MiniZinc language here, the coverage of the language is far from exhaustive. For those new to MiniZinc, Coursera offers an introductory course that can be found here: https://www.coursera.org/learn/basic-modeling. *A* rather concise, but nonetheless useful tutorial can be found here: https://www.MiniZinc.org/tutorial/MiniZinc-tute.pdf. My aim here is rather to solve what I view to be interesting problems, employing where possible, only the more elementary features of MiniZinc. A challenge one faces when learning or applying any programming language is to have an ample supply of accessible code samples upon which to study and build

upon. I believe code samples are even more vital when programming in a functional styled language like MiniZinc. Whereas no more than few examples of loop constructs or conditional branch statements might suffice when learning to program in Java or Python, the same is not true when first approaching a functional or functional styled language like Haskell, R, or MiniZinc. Learning to program functionally is best supported through accessible and at the same time intriguing code samples. Unfortunately it has been my experience that many sample MiniZinc models, due to the complexity of the problem being addressed, tend to obscure the language features employed. This makes it less likely for one new to the language to be successful later in applying what has been presented in a novel setting. Hopefully, the nature of the problems solved here in MiniZinc will serve as a compendium of readily accessible examples, which one can refer to in order to adapt or extend.

In the following pages, when it is possible to do so, problems are categorized by the language features employed in the problem's solution. This is opposed to the usual approach of a grouping by problem content. In this way, a MiniZinc modeler when needing a particular language feature will hopefully be able to index one or more relevant solved examples. This approach, of course, is the one familiarly taken by dictionaries and lexicons where sample sentences illustrate a term's usage. The tract's Contents section given below can then be seen to be organized in this manner.

A constraint solver like MiniZinc is a powerful tool. As one becomes more proficient in its use, one's appreciation of scope and range of problems amenable to being solved with the language will surely grow. In truth, one could possibly be lulled into adopting the (unrealistic) mindset that every problem encountered can be viewed as a constraint problem solvable by MiniZinc. Acquiring such narrow

view should not raise any alarm for the liberally educated who will no doubt classify MiniZinc as just another problem solving tool in their kit.

I should add that when time and energy permits, I encourage the reader to attempt to solve a given problem before referencing the given MiniZinc solution. One should realize that for almost all the problems presented here, a purely mathematical solution exists, which for the so inclined can be of equal interest to discover.

Before proceeding to the problems and their solution in MiniZinc, I will point out that all the MiniZinc code presented here is linked to authors GitHub account, which can be found here https://github.com/pjoscely.

Contents

Modeling with Integer Variables ..... page 7

Modeling with Float Variables ..... page 11

Modeling with Conditionals

Modeling with Arrays

Modeling with Sets

Modeling with Predicates

Modeling with Functions

Modeling with Integer Variables

For our initial example, let us consider the following high school algebra coin problem: https://github.com/pjoscely/Math-Comp-MiniZinc/blob/master/AMC%208/Alg%202%20H%20Coin%20Problem.mzn

> A bag contains twice as many pennies as nickels and four more dimes than quarters. Find all possibilities for the number of each coin if their total value is $2.01.

```
%Coin variables
var 0..201:P;
var 0..41:N;
var 0..21:D;
var 0..8:Q;

%twice as many pennies as nickels
constraint P = 2*N;

%four more dimes than quarters
constraint D = Q + 4;

%total value is $2.01
constraint 201 = P + 5*N + 10*D + 25*Q;

%required solve statement
solve satisfy;

%display results
output["P = "++"\(P)"++" N = "++"\(N)"++" D = "++"\(D)"++" Q = "++"\.
(Q)"];

/*
P = 46 N = 23 D = 4 Q = 0
----------
P = 36 N = 18 D = 5 Q = 1
----------
P = 26 N = 13 D = 6 Q = 2
----------
P = 16 N = 8 D = 7 Q = 3
----------
P = 6 N = 3 D = 8 Q = 4
----------
==========
*/
```

The program's output/solution is included as a comment after the code. Single line comments in MiniZinc begin with a "%", while multiline comments begin with "/*" and end with "*/". The double dashed line concluding the output's indicates

that MiniZinc found no more solutions. While the model is self-explanatory, I should be pointed out that the defined decision variables are declared with integer bounds in accordance with the problem. Doing so, narrows the search and lessens the final number of constraint statements. More on this later.

The following is problem 7 from the 2019 American Mathematics Competition 8 Examination. https://github.com/pjoscely/Math-Comp-MiniZinc/blob/master/AMC%208/amc8_2019_7.mzn

```
Shauna takes five tests, each worth a maximum of 100 points.
Her scores on the first three tests are 76, 94, and 87.
In order to average 81 for all five tests, what is the lowest score
she could earn on one of the other two tests?

(A)  48  (B)  52  (C)  66  (D) 70  (E) 74


%test scores
par int: first = 76;
par int: second = 94;
par int: third = 87;

var int: fourth;
var int: fifth;

%bounds on possible test scores
constraint fourth >= 0;
constraint fifth >= 0;
constraint fourth < 101;
constraint fifth < 101;

%final average constraint
constraint (first + second + third + fourth + fifth)/5 = 81;

%minimize fourth test score
solve minimize fourth;

%display result
output [" lowest fourth score = "++"\(fourth)"]

 %lowest score = 48
```

The model above introduces the shorthand “par” for fixed a parameter declaration. The decision variables fourth and fifth are declared as “int” and are constrained later in the model. As with the model presented in the Preface, this

model introduces the very useful “solve minimize” command, which has its counterpart “solve maximize”.

A small-sized knapsack problem is problem 2 from the 2018 American Mathematics Competition 12A Examination. https://github.com/pjoscely/Math-Comp-MiniZinc/blob/master/AMC%2012/amc12A_2018_2.mzn

```
While exploring a cave, Carl comes across a collection of 5-pound rocks
worth 14 each, 4-pound rocks worth 11 each, and 1-pound rocks worth 2
each. There are at least 20 of each size. He can carry at most 18 pounds.
What is the maximum value, in dollars, of the rocks he can carry out
of the cave?

(A) 48    (B) 49     (C) 50     (D) 51    (E) 52

%decision variables
var int:five_p;
var int:four_p;
var int:one_p;

%positive constraints
constraint five_p >=0;
constraint four_p >=0;
constraint one_p >=0;

%define a wt variable
var int:wt = 5*five_p+4*four_p+one_p;

%carry at most 18 pounds
constraint wt<=18;

%maximum value, in dollars
solve maximize 14*five_p+11*four_p+2*one_p;

%display the result
output["five_p = "++"\(five_p)"++" four_p = "++"\(four_p)"++" one_p = "+
+"\.
(one_p)"++ " Max value = "++"\(wt)"];

/*
five_p = 2 four_p = 2 one_p = 0 Max value = 18
==========
*/
```

MiniZinc found the maximum value of 18 is obtained by placing two 5-pound rocks and two 4-pound rocks in the knapsack. Although it is customary to place constraints after variable definitions, this is not required by MiniZinc as can be seen

by the location within the code of the definition of the “wt” variable. Lastly, the three positive constraints insures the model will terminate.

Problem 21 from the 2018 American Mathematics Competition 8 Examination demonstrates the use of MiniZinc’s mod function. https://github.com/pjoscely/Math-Comp-MiniZinc/blob/master/AMC%208/amc8_2018_21.mzn

```
Problem 21 AMC 8 2018
How many positive three-digit integers have a remainder of 2
when divided by 6, a remainder of 5 when divided by 9,
and a remainder of 7 when divided by 11?

(A) 1  (B)  2  (C)  3  (D)  4  (E)  5

%possible positive three-digit integers
var 100..999:n;

%remainder constraints
constraint n mod 6 = 2;
constraint n mod 9 = 5;
constraint n mod 11 = 7;

solve satisfy;

%display result
output ["n = "++"\(n)"]

/*
n = 194;
----------
n = 392;
----------
n = 590;
----------
n = 788;
----------
n = 986;
----------
*/
```

Modeling with Float Variables

MiniZinc has the capability of handling "real number" constraint solving using floating point solving. For the simple LP problem below, one needs to change the default configuration. If using the MiniZinc IDE this is accomplished by opening the Configuration tab and under solving select G12 MIP. https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/LP1.mzn

```
Find the maximal value of z = 3x + 4y
subject to the following constraints:
x + 2y <= 14, 3x - y >= 0, x - y <= 2

%define float variables
var float:x;
var float:y;

%constraints
constraint x+2*y<=14;
constraint 3*x-y>=0;
constraint x-y<=2;

%maximize
solve maximize 3*x+4*y;

%display the result
output["x = "++"\(x)"++" y = "++"\(y)"++" Max = "++"\(3*x+4*y)"];

%x = 6.0 y = 4.0 Max = 34.0
```

Consider now the following Fractional Knapsack problem. In this case, items can be broken into smaller pieces, hence we can select fractions of items.

Let us suppose that the capacity of the knapsack is $W = 60$ and the list of provided items are shown in the following table:

https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_fractional_knapsack.ht

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24

Like the LP model, this model requires us to select under solving the G12 MIP option. https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/frac_knapsack.mzn

```
%define float variables
var 0.0..1.0:A;
var 0.0..1.0:B;
var 0.0..1.0:C;
var 0.0..1.0:D;

%sack can contain no more than 60
constraint 40*A+10*B+20*C+24*D<=60;

%maximize profit
solve maximize 280*A+100*B+120*C+120*D;

%display the result
output["A = "++show_float(1, 2, A)++" B = "++show_float(1, 2, B)++
        " C = "++show_float(1, 2, C)++" D = "++show_float(1, 2, D)++
        " Max = "++show_float(5, 2,280*A+100*B+120*C+120*D)];

%A = 1.00 B = 1.00 C = 0.50 D = 0.00 Max = 440.00
```

Minizinc returns the optimal solution of one unit of A and B and a half unit of C. D is not taken. The Fractional Knapsack problem may be solved with a greedy algorithm in polynomial time whereas the classic knapsack problem is NP-hard. These issues, although vitally important for the imperative programmer working in say Java, are not transparent when using MiniZinc.

Nonlinear problems like those encountered when studying the technique of Lagrange Multipliers can sometimes naively solved with MiniZinc, provided initial bounds are put on the decision variables. Below is a problem usually handled by the using Lagrange Multipliers. Interestingly, this model fails under the G12 MIP

option, but yields a solution if the default Geocode(bundled) option is selected.

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/lagrange1.mzn

Maximize $81x^2 + y^2$ subject to the constraint $4x^2 + y^2 = 9$

```
%define float variables
var -1.5..1.5:x;
var -3.0..3.0:y;

constraint 4*x*x+y*y = 9;

solve maximize 81*x*x+y*y;


%display the result
output["x = "++show_float(6, 1, x)++" y = "++show_float(6, 2, y)++
      " Max = "++show_float(6, 2, 81*x*x+y*y)];

%x =   -1.5 y =  -0.00 Max = 182.25
```

A good practice is to always bound decision variables where possible. For this model, bounding the variables is actually required for the model to terminate. This is often the case when using float variables. Since all variables are squared, the additional solution x = 1.5 y = -0.00 is easily obtained by inspection.

Contrast this to the problem below, which Minizinc fails to return a solution in any reasonable time.

Minimize $x^2 + 2y^2 - 4y$ subject to the constraint $x^2 + y^2 = 9$

```
%define float variables
var -3.0..3.0:x;
var -3.0..3.0:y;

constraint x*x+y*y = 9;

solve m x*x +2*y*y-4*y;

%display the result
output["x = "++show_float(6, 1, x)++" y = "++show_float(6, 2, y)++
      " Max = "++show_float(6, 2, x*x +2*y*y-4*y)];
```