# Elementary MiniZinc for Mathematics and Puzzles

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
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* somewhat concise, but nonetheless useful tutorial can be found here: https://www.MiniZinc.org/tutorial/MiniZinc-tute.pdf. My aim here rather is 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, problems are categorized by the language features employed in the problem's solution. This is opposed to the common grouping by problem content. In this way, a MiniZinc modeler when needing a particular language feature will be able to more easily index one or more relevant solved examples. This approach, of course, is the one familiarly taken by many programming language expositions as a dictionaries and lexicons where sample sentences illustrate a term's usage.  The 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 a narrow view should not raise any alarm for the liberally educated who will no doubt consign MiniZinc to be just another problem solving tool in their kit.

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

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

```
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, one selects 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
```

The maximal solution is supplied as a comment on the program's last line.

Matt Parker, the "Stand-Up Mathematician", features a bi-monthly puzzle on his Thinks Maths website. His puzzle #11: https://www.think-maths.co.uk/agepuzzle is given below. A MiniZinc model solution, which employs floating point variables follows the puzzle statement. Again should the reader wish to run this model the G12 MIP option should be selected upon opening the Configuration tab:

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

```
David and Anton's ages combined equals 65. David is currently three times as
old as Anton was when David was half as old as Anton will be when Anton is
three times as old as David was when David was three times as old as Anton.

Puzzle for submission: How old is David?

% David's ages
var float :D0;
var float :D1;
var float :D2;
var float :D3;

% Anton's ages
var float :A0;
var float :A1;
var float :A2;
var float :A3;

% ages combined equals 65
constraint D0+A0 = 65;

% David is currently three times as old as Anton was when
constraint D0 = 3*A1;

% David was half as old as Anton will be when
constraint D1 = (0.5)*A2;

% Anton is three times as old as David was when
constraint A2 = 3*D3;

% David was three times as old as Anton
constraint D3 = 3*A3;

% constant difference in ages
% across time instances
constraint D1-D0 = A1-A0;
constraint D2-D1 = A2-A1;
constraint D3-D2 = A3-A2;

solve satisfy;

output
[
  "David age's: " ++ show(D0)
];

%David age's: 37.5
```

As is the case for many examples, once the appropriate variables have been defined, a problem statement has a seemingly direct translation into MiniZinc constraints. David's age is supplied as a comment on the program's last line.

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 two floating point models above, 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. AS MiniZinc relies on standard techniques like backtracking and constraint programming the user should keep in mind that MiniZinc is not a panacea. Like any other specially crafted imperative program in the absence of an efficient algorithm,  a MiniZinc program's performance will suffer when negotiating very large search spaces.

The formula below is used to calculate the fixed monthly payment (P) required to fully amortize a loan of L dollars over a term of n months at a monthly interest rate of c. [If the quoted rate is 6%, for example, c is .06/12 or .005].

$$P = \frac{L[c((1+c)^n]}{[(1+c)^n - 1]}$$

The Minizinc code below: https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/mortgage.mzn calculates the allowable home loan given a set monthly payment, interest rate, and time period. Since the power function "pow" apparently does not allow for decision variables, the formula is used here only to find the allowable loan. It should be pointed out that MiniZinc allows for the easy creation of a separate data file containing given parameter values. Details on how to accomplish this can be found here: https://www.MiniZinc.org/tutorial/MiniZinc-tute.pdf.

```
%Monthly payments
par float:P = 1200.0;

%Interest rate divided over 12
par float:c = 0.002417; % 2.9 percent divided by 12

%30 years times twelve months
par float:n = 360.0;

%Loan allowed
var float:L;
```

```
%Mortgage equation

constraint P = L*(c*pow(1+c,n))/(pow(1+c,n)-1);

solve satisfy;

%display the result
output["Maximum loan = $"++show_float(6, 2, L)];

Maximum loan = $288,287.32
```

A maximum loan of $288,287.32 can be had for monthly payments of $1200, amortized monthly over 30 years, at an interest rate of 2.9%.

Nonlinear optimization problems can sometimes be solved naively with MiniZinc, provided initial bounds are put on the decision variables. Below is a problem that is usually handled with 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
```

When using MiniZinc, it is a good practice to bound decision variables when possible. For this model, bounding the variables is actually required for the model

to terminate. Since all variables are squared, the additional solution x = 1.5 y = -0.00 is easily obtained by inspection.

Although, it is not entirely our purpose here to categorize problems, which can or cannot be solved with MiniZinc, it is interestingly to note that the similar problem below 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)];
```

## Modeling with Conditionals

Like other programming languages MiniZinc supports the formation of Boolean statements. These are indispensable when tailoring constraints for a particular model. The usual “and”, “or”, and “not” from say Python are expressed in MiniZinc as “/\”, “\/“, and “!” respectively.

For our first example, we employ the Boolean not “!” operator in MiniZinc, to solve a simple map coloring problem. Here we color the 6 New England States with three colors, the colors are represented by the numbers 1, 2, 3. This example is adapted from: https://www.minizinc.org/doc-2.4.3/en/modelling.html#sec-modelling

```
% Coloring New England using n colors
int: n = 3;

% State variables
var 1..n: Maine;
var 1..n: NewHamp;
var 1..n: Vermont;
var 1..n: Conn;
var 1..n: Mass;
var 1..n: RhodeI;

% bordering states have different colors
constraint Maine != NewHamp;
constraint NewHamp != Vermont;
constraint NewHamp != Mass;
constraint Vermont != Mass;
constraint Mass != Conn;
constraint Mass !=  RhodeI;
constraint Conn != RhodeI;

solve satisfy;

% display map coloring
output ["Maine = \(Maine)\t New Hamp = \(NewHamp)\t Vermont = \( Vermont)
\n”, "Conn = \(Conn)\t Mass = \(Mass)\t RhodeI = \(RhodeI)\n”, “\n"];
```

```
Maine = 1       New Hamp = 2       Vermont = 3
Conn = 3        Mass = 1           RhodeI = 2
```

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

Opening the Configuration tab and changing the under user defined behavior to display all solutions, Minizinc displays 24 distinct colorings. If only two colors are used then MiniZinc outputs that the problem is unsatisfiable.

For the next example, consider the July 2018 IBM Ponder This Challenge:

*Let's call a triplet of natural numbers "obscure" if one cannot uniquely deduce them from their sum and product. For example, {2,8,9} is an obscure triplet, because {3,4,12} shares the same sum (19) and the same product (144). Find a triplet of ages {a,b,c} that is obscure and stays obscure for three more years: {a+1,b+1,c+1}, {a+2,b+2,c+2} and {a+3,b+3,c+3}.*

*https://www.research.ibm.com/haifa/ponderthis/challenges/July2018.html*

```
% Decision variables
var 1..120: a;
var 1..120: b;
var 1..120: c;

var 0..120: x1;
var 0..120: x2;
var 0..120: x3;

var 0..120: y1;
var 0..120: y2;
var 0..120: y3;

var 0..120: z1;
var 0..120: z2;
var 0..120: z3;

var 0..120: w1;
var 0..120: w2;
var 0..120: w3;

% Required constraints
constraint a+b+c = x1+x2+x3;
constraint a*b*c = x1*x2*x3;
constraint a != x1 \/ b != x2 \/c != x3;
constraint a <= b /\ b <= c;
constraint x1 <= x2 /\ x2 <= x3;

constraint a+b+c+3 = y1+y2+y3;
constraint (a+1)*(b+1)*(c+1) = (y1)*(y2)*(y3);
constraint a+1 != y1 \/ b+1 != y2 \/ c+1 != y3;
constraint y1 <= y2 /\ y2 <= y3;

constraint a+b+c+6 = z1+z2+z3;
constraint (a+2)*(b+2)*(c+2) = (z1)*(z2)*(z3);
constraint a+2 != z1 \/ b+2 != z2 \/ c+2 != z3;
```

```
constraint z1 <= z2 /\ z2 <= z3;

constraint a+b+c+9 = w1+w2+w3;
constraint (a+3)*(b+3)*(c+3) = (w1)*(w2)*(w3);
constraint a+3 != w1 \/ b+3 != w2 \/ c+3 != w3;
constraint w1 <= w2 /\ w2 <= w3;

solve satisfy;
```
https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/JulyIBM.mzn

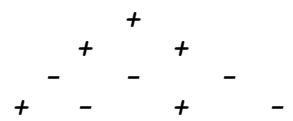
One of the many solutions found by this MiniZinc model is given in the table below:

| obscure triplet | sum | product | shared triplet |
|---|---|---|---|
| {15, 22, 48} | 85 | 15840 | {12, 33, 40} |
| {16, 23, 49} | 88 | 18032 | {14, 28, 46} |
| {17, 24, 50} | 91 | 20400 | {20, 20, 51} |
| {18, 25, 51} | 94 | 22950 | {15, 34, 45} |

This IBM puzzle perhaps has some intrinsic interest, but I would imagine that a reader relatively new to MiniZinc might be more interested in the use of the language's boolean operators in composing constraints. The model's constraint code is redundant, and the copious use of similarly named variables makes one wonder if arrays exist in MiniZinc. While the question of arrays will be taken up in the next section, for now the model, in the spirit of the this tract, is presented as is. Should the reader wish to run this model the default Geocode(bundled) option should be selected.

Along with Boolean statements, MiniZinc allows for conditional branch statements like those found in imperative programming languages. For an example, let us consider problem 18 from the 2018 American Mathematics Competition 8 Examination: https://artofproblemsolving.com/wiki/index.php/2018_AMC_8_Problems/Problem_19#Problem_19

*In a sign pyramid a cell gets a "+" if the two cells below it have the same sign, and it gets a "-" if the two cells below it have different signs. The diagram below illustrates a sign pyramid with four levels. How many possible ways are there to fill the four cells in the bottom row to produce a "+" at the top of the pyramid?*



```
% Model + by 0 - by 1
var 0..1:row11;

var 0..1:row21;
var 0..1:row22;

var 0..1:row31;
var 0..1:row32;
var 0..1:row33;

var 0..1:row41;
var 0..1:row42;
var 0..1:row43;
var 0..1:row44;

% Initialize the first row
constraint row11 = 0;

% Pyramid cell constraints
constraint if row21 = row22 then row11 = 0 else row11 = 1 endif;
constraint if row31 = row32 then row21 = 0 else row21 = 1 endif;
constraint if row32 = row33 then row22 = 0 else row22 = 1 endif;
constraint if row41 = row42 then row31 = 0 else row31 = 1 endif;
constraint if row42 = row43 then row32 = 0 else row32 = 1 endif;
constraint if row43 = row44 then row33 = 0 else row33 = 1 endif;
solve satisfy;

output["\(row41)"++" "++"\(row42)"++" "++"\(row43)"++" "++"\(row44)"];
```

```
0 0 0 0
1 0 0 1
1 1 0 0
0 1 0 1
1 0 1 0
0 0 1 1
0 1 1 0
1 1 1 1
```

As can be seen above the model finds a total of 8 ways to fill the bottom row and satisfy the problem's requirements. Here the + sign is represented by a 0 and the - sign is represented by a 1 in the model. The syntax of MiniZinc's "if then endif"

construct can be gleaned from its use above. Should the reader need more details, on “if”, nested “if” statements, and other conditional statements in MiniZinc, they can be found here: https://www.minizinc.org/doc-2.4.3/en/modelling2.html#conditional-expressions.

## Modeling with Arrays

As was seen, for some of the the models developed above the number of variables employed became quite large. The more complex the problem one is faced with, the more likely it is that one will require more than a modicum of variables and consequent constraints to achieve a solution. As with imperative programming languages, MiniZinc allows for the creation of data structures such as arrays and sets and their incorporation in constraints to help manage these situations. In fact, as is noted in the MiniZinc tutorial, one is often "interested in building models where the number of constraints and variables is dependent on the input data". Arrays and sets will also allow us this flexibility to model variable size inputs.

For those already accustomed to arrays in imperative languages, MiniZinc presents little in the way that is new or different. Suffice it to say that arrays in MiniZinc mirror those found in imperative languages, excepting that array indices start with one instead of zero. For our next MiniZinc array example, let us consider problem 25 from the 2017 American Mathematics Competition 10B Examination:

*Last year Isabella took 7 math tests and received 7 different scores, each an integer between 91 and 100, inclusive. After each test she noticed that the average of her test scores was an integer. Her score on the seventh test was 95. What was her score on the sixth test?*

*(A) 92 (B) 94 (C) 96 (D) 98 (E) 100*

```
include "alldifferent.mzn";

% array for possible test scores
array[1..7] of var 91..100: d;

%After each test the average of the test scores is an integer.
constraint (d[1]+d[2]) mod 2 = 0;
constraint (d[1]+d[2]+d[3]) mod 3 = 0;
constraint (d[1]+d[2]+d[3]+d[4]) mod 4 = 0;
constraint (d[1]+d[2]+d[3]+d[4]+d[5]) mod 5 = 0;
constraint (d[1]+d[2]+d[3]+d[4]+d[5]+d[6]) mod 6 = 0;
constraint (d[1]+d[2]+d[3]+d[4]+d[5]+d[6]+d[7]) mod 7 = 0;

%Her score on the seventh test was 95
constraint d[7] =  95;
```

```
%7 different scores in all
constraint alldifferent(d);

solve satisfy;

% display scores
output["6th score: "++"\(d[6])"++" "++" Scores: "++"\(d)"];
```

```
6th score: 100  Scores: [93, 91, 92, 96, 98, 100, 95]
6th score: 100  Scores: [96, 92, 91, 93, 98, 100, 95]
6th score: 100  Scores: [96, 92, 91, 97, 94, 100, 95]
6th score: 100  Scores: [96, 92, 97, 91, 94, 100, 95]
6th score: 100  Scores: [91, 93, 92, 96, 98, 100, 95]
6th score: 100  Scores: [92, 96, 91, 93, 98, 100, 95]
6th score: 100  Scores: [92, 96, 91, 97, 94, 100, 95]
6th score: 100  Scores: [92, 96, 97, 91, 94, 100, 95]
```

MiniZinc found a total of 8 solutions, which are listed after the model's code. Later we will see that it is possible to compress the clumsy array sums in the constraint equations, but for now they are left to serve as examples of MiniZinc's array syntax. It should be noted that the include "alldifferent.mzn" statement allows the formation of the "alldifferent" global constraint as required by the problem. This is probably one of the most useful global constraints MiniZinc offers the modeler.

The next example demonstrates the ease with which MiniZinc can solve the interesting, albeit challenging combinatorial problem: #11 from the from the 2007 American Mathematics Competition 10A Examination:

*The numbers from 1 to 8 are placed at the vertices of a cube in such a manner that the sum of the four numbers on each face is the same. What is this common sum?*

*A)  14  (B)  16  (C)  18  (D)  20  (E)  24*

```
include "alldifferent.mzn";

% array for possible vertices values
array[1..8] of var 1..8: a;

%common sum of the four numbers on each face
constraint a[1]+a[2]+a[3]+a[4] = a[7]+a[2]+a[3]+a[6];
constraint a[7]+a[2]+a[3]+a[6] = a[1]+a[5]+a[8]+a[4];
constraint a[1]+a[5]+a[8]+a[4] = a[6]+a[5]+a[8]+a[7];
constraint a[6]+a[5]+a[8]+a[7] = a[1]+a[2]+a[6]+a[5];
constraint a[1]+a[2]+a[6]+a[5] = a[7]+a[8]+a[3]+a[4];
```

```
%6 different numbers
constraint alldifferent(a);

solve satisfy;
output["common sum = “++”\(a[1]+a[2]+a[3]+a[4])"];
```

MiniZinc found a total of 144 different assignments of the numbers 1 to 8 to the vertices of the cube with a constant face sum. Each of these assignments have the common sum of 18.

For our next MiniZinc array example, let us consider problem 25 from the 2019 American Mathematics Competition 10B Examination: https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%2010/2019_AMC_10B_25.mzn

```
How many sequences of 0 and 1 of length 19 are there that
begin with a 0, end with a 0, contain no two consecutive 0s,
and contain no three consecutive 1s?

A    55    B    60    C 65    D    70    E    75

% use an array to model sequences
array[1..19] of var 0..1: s;

% all sequences begin with 0
constraint s[1] = 0;

% all sequences end with 0
constraint s[19] = 0;

% contain no two consecutive 0s
constraint forall(i in 1..18)(s[i]=1 \/ s[i+1] = 1);

% contain no three consecutive 1s
constraint forall(i in 1..17)(s[i] = 0 \/ s[i+1] = 0 \/ s[i+2] = 0);

solve satisfy;

% 65 solutions
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0])
[0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0])
[0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0])
. . .
[0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0])
[0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0])
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0])
```

MiniZinc found 65 solutions, for which the first and last three are listed above. Often, as with this problem, the total number of solutions are asked for instead of a complete listing of solutions. If one opens the configuration tab and under

advanced options selects output statistics for solving then MiniZinc will additionally display the number of solutions found. Even after selecting this option, occasionally MiniZinc will fail to display the total number of solutions. In this case, one can infer the total number from the tally MiniZinc displays.

This model introduces the “forall( )” statement. Just as “for loops” go hand in hand with arrays when programming in imperative languages, the “forall( )” statement is indispensable in forming constraints with arrays in MiniZinc.

MiniZinc also allows the modeler the ability to create enumerated types. As with the enumerated types found in imperative languages, an enumerated type of size n in MiniZinc behaves much like the integers 1, 2, 3 . . . n. They can be compared by the order they appear in the definition, act as indices of an array, and be used anywhere in the code where an integer can be used.

For our first example of enumerated types in MiniZinc, we consider another Matt Parker puzzle: https://www.think-maths.co.uk/catsanddogs.

```
How many ways can you completely fill your ten kennels, using only cats
and dogs (one animal per kennel), such that no two cats are in adjacent
kennels?

% enumerated data type
enum pets = {cat,dog};

% kennel of size 10
array[1..10] of var pets: kennel;

% no adjacent cats
constraint forall(i in 1..9)((kennel[i] = cat) -> (kennel[i+1] != cat));

solve satisfy;

% 144 solutions

[cat, dog, cat, dog, cat, dog, cat, dog, cat, dog])
[dog, dog, cat, dog, cat, dog, cat, dog, cat, dog])
[dog, cat, dog, dog, cat, dog, cat, dog, cat, dog]);
. . .
```

```
[dog, dog, dog, dog, dog, dog, dog, dog, dog, cat])
[cat, dog, dog, dog, dog, dog, dog, dog, dog, dog])
[dog, dog, dog, dog, dog, dog, dog, dog, dog, dog])
```

MiniZinc found 144 total solutions of which the first and last three are shown above. The enumerated type “cat” and “dog” clearly could be represented by a pair of integer values, but their use here allows the model to express more transparently how the puzzle is modeled.

Before moving on to sets in MiniZinc, we consider model problem 4 from the 2018 American Mathematics Competition 10A Examination. The MiniZinc model presented here illustrates the use of multiple constraints, each using an implies (—>) construct. The last constraint employs the “sum” array function. The sum function is one among other functions that are available in MiniZinc to operate on the arrays: https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%2012/amc12A_2018_4.mzn

```
How many ways can a student schedule 3 mathematics courses --
algebra, geometry, and number theory -- in a 6-period day if
no two mathematics courses can be taken in consecutive periods?

(What courses the student takes during the other 3 periods
is of no concern here.)

(A)3     (B)6       (C)12      (D)18       (E) 24
```

```
% D is a dummy or filler course
enum c = {A,G,N,D};

array[1..6] of var c: s;

% no two mathematics courses can be taken in consecutive periods
constraint forall(d in 1..5)((s[d] = A) -> (s[d+1] != A));
constraint forall(d in 1..5)((s[d] = A) -> (s[d+1] != G));
constraint forall(d in 1..5)((s[d] = A) -> (s[d+1] != N));

constraint forall(d in 1..5)((s[d] = G) -> (s[d+1] != G));
constraint forall(d in 1..5)((s[d] = G) -> (s[d+1] != A));
constraint forall(d in 1..5)((s[d] = G) -> (s[d+1] != N));

constraint forall(d in 1..5)((s[d] = N) -> (s[d+1] != N));
constraint forall(d in 1..5)((s[d] = N) -> (s[d+1] != A));
constraint forall(d in 1..5)((s[d] = N) -> (s[d+1] != G));
```

```
% a math course may be scheduled only once
constraint forall(i in 1..5, j in 2..6 where i <j)((s[i] = A) -> (s[j] !=
A));
constraint forall(i in 1..5, j in 2..6 where i <j)((s[i] = G) -> (s[j] !=
G));
constraint forall(i in 1..5, j in 2..6 where i <j)((s[i] = N) -> (s[j] !=
N));

constraint sum(i in 1..6)(bool2int(s[i] = D)) = 3;

solve satisfy;
% display schedules
output[show(s)];

%solutions:     24

[N, D, G, D, A, D]
[G, D, N, D, A, D]
[G, D, D, A, D, N]

[A, D, G, D, D, N]
[G, D, N, D, D, A
[N, D, G, D, D, A]
```

MiniZinc found 24 total solutions for which the first and last three are shown above. We note here again the convenience of the enumerated data type. The use of bool2int (boolean to integer) function in conjunction with the sum function insures exactly three mathematics (or equivalently three filler) courses are scheduled. This problem is an example of a very small size scheduling problem. Scheduling is of vital importance in so many disciplines. The MiniZinc language is ideally suited for modeling scheduling and related problems.

For your last example, we consider Problem 5 from the 2020 American Mathematics Competition 12A Examination.

*The 25 integers from -10 to 14 inclusive, can be arranged to form a 5-by-5 square in which the sum of the numbers in each row, the sum of the numbers in each column, and the sum of the numbers along each of the main diagonals are all the same.*

*What is the value of this common sum?*

*A 2    B 5    C 10    D 25    E 50*

```
include "alldifferent.mzn";

% 5 x 5 2-D array
array[1..5, 1..5] of var -10..14: g;
var int:c;

%Diagonal conditions
constraint sum (i in 1..5) (g[i,i]) = c;
constraint sum (i in 1..5) (g[i,6-i]) = c;

%Row conditions
constraint sum (j in 1..5) (g[1,j]) = c;
constraint sum (j in 1..5) (g[2,j]) = c;
constraint sum (j in 1..5) (g[3,j]) = c;
constraint sum (j in 1..5) (g[4,j]) = c;
constraint sum (j in 1..5) (g[5,j]) = c;

%Column conditions
constraint sum (i in 1..5) (g[i,1]) = c;
constraint sum (i in 1..5) (g[i,2]) = c;
constraint sum (i in 1..5) (g[i,3]) = c;
constraint sum (i in 1..5) (g[i,4]) = c;
constraint sum (i in 1..5) (g[i,5]) = c;

% insure all 25 numbers used exactly once
constraint alldifferent([g[i,j]|i in 1..5, j in 1..5]);

solve satisfy;

[14, -7, 12, -8, -1, -6, 10, -3, 8, 1, -2, 7, -10, 2, 13, -5, 4, 0, 5, 6,
9, -4, 11, 3, -9]

c = 10
```

MiniZinc found a plethora of solutions to this problem. Only one is listed above, along with the common sum value: “c = 10”. This model employs two-dimensional arrays, which is another useful data structure available in MiniZinc that is universally found in other programming languages.

## Modeling with Sets

Historically, the concept of a set has been fundamental to mathematics. Sets afford a unifying language for both mathematical definitions and analysis. In accordance, modern languages like Java and Python allow for the formation and manipulation of sets of various data types. MiniZinc singularly allows for sets containing integers to be decision variables. This ability to form constraints on integer valued sets is yet another powerful modeling feature of MiniZinc.

For our first example of modeling with sets, we consider problem 11 from the 1991 American Junior High School Examination: https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%208/AJHSME_1991_11.mzn

```
There are several sets of three different numbers whose sum is 15
which can be chosen from {1,2,3,4,5,6,7,8,9}.
How many of these sets contain a 5?

(A)3   (B)  4   (C) 5   (D) 6   (E) 7

% form base set
set of int: possible = {1,2,3,4,5,6,7,8,9};

% form all subsets of possible
var set of possible:c;

% sets of three different numbers
constraint card(c)=3;

% sum is 15
constraint sum (i in c) (i) = 15;

% sets contain a 5
constraint 5 in c;

solve satisfy;

c = {1,5,9}
c = {2,5,8}
c = {3,5,7}
c = 4..6;
```

This code above shows how sets are formed in MiniZinc. The card() function is one among others of set operations available for use in constraints. Set membership is expressed with the “in” statement. Lastly, MiniZinc found four

solutions, which are listed after the code. Most likely since MiniZinc represents sets in terms of arrays, the set {4, 5, 6} is displayed as the array 4..6 in the last line.

Similar to the above problem is problem 9 taken from the vast collection of challenging problems provided by Project Euler: https://projecteuler.net/

```
Special Pythagorean triplet

Problem 9
A Pythagorean triplet is a set of three natural numbers, a < b < c, for
which, a^2+ b^2 = c^2.

For example, 3^2 + 4^2 = 9 + 16 = 25 = 5^2..

There exists exactly one Pythagorean triplet for which a + b + c = 1000.

Find the product abc.


% restrict search
var 1..1000:a;

var int: b;

var int:c;

% b, c depend on a
constraint b in a+1..1000;

constraint c = 1000 - a - b;

% triangle conditions
constraint a < b + c /\ b < a + c /\ c < a + b;

% Pythagorean condition
constraint a*a+b*b=c*c;

% perimeter condition
constraint a+b+c=1000;

solve satisfy;

% display solution
output["product abc = "++"\(a*b*c)"];


product abc = 31875000
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/proj_euler_9.mzn

Another set constraint example is problem 12 from the 2004 American Mathematics Competition 10A Examination: https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%2010/2004_AMC_10A_12.mzn

*Henry's Hamburger Heaven offers its hamburgers with the following condiments: ketchup, mustard, mayonnaise, tomato, lettuce, pickles, cheese, and onions. A customer can choose one, two, or three meat patties, and any collection of condiments.*
*How many different kinds of hamburgers can be ordered?*

*(A) 24 (B) 256 (C) 768 (D) 40,320 (E) 120,960*

```
%ingredients: h1, h2, h3, 1, 2, or 3 patties
enum h = {h1, h2, h3, ketchup, mustard, mayonnaise, tomato, lettuce,
pickles, cheese, onions};

%all possible subsets of h
var set of h: possible;

%must have meat patty
constraint card(possible intersect {h1, h2, h3}) = 1;

solve satisfy;

% display sets
output["A burger = "++"\(possible)"];
```

```
768 solutions

A burger = {h1, ketchup, mustard, mayonnaise, tomato, lettuce, pickles,
cheese, onions}

A burger = {h1, ketchup, mustard, mayonnaise, tomato, lettuce, pickles,
cheese}

A burger = {h1, ketchup, mustard, mayonnaise, tomato, lettuce, pickles,
onions}
. . .
A burger = {h3, cheese}

A burger = {h3, onions}

A burger = {h3}
```

MiniZinc found 768 total solutions for which the first and last three are shown above. The enumerated data type is used to specify a hamburger's ingredients. Note the "intersect" (intersection of two sets) along with the "card" function constrains any solution set to have exactly one, two, or three hamburger patties. Since there is only one constraint on possible subsets of {h1, h2, h3, ketchup,

mustard, mayonnaise, tomato, lettuce,  pickles, cheese, onions} the large number of solutions (768) is not surprising.

Below is yet another Matt Parker puzzle that was ideally suited for solving with sets in MiniZinc: https://www.think-maths.co.uk/primepairs

*Rearrange the numbers from 1-9, such that all adjacent pairs sum to a prime number.*

```
include "alldifferent.mzn";


% the numbers from 1-9
par int:n = 9;

% set of possible prime sums
par set of int: primes = {2,3,5,7,11,13,17,19};

% represent solutions in an array
array[1..n] of var 1..n: s;

% adjacent pairs sum to a prime number
constraint forall(i in 1..n-1)((s[i]+s[i+1]) in primes);

% insure each digit is used exactly once
constraint alldifferent(s);

solve satisfy;

% display solutions
output["\(s)"];

% solutions:     140

[7, 4, 9, 8, 5, 6, 1, 2, 3]
[7, 4, 3, 8, 5, 6, 1, 2, 9]
[7, 4, 1, 6, 5, 8, 3, 2, 9]
. . .
[7, 6, 5, 2, 1, 4, 9, 8, 3]
[3, 2, 1, 6, 7, 4, 9, 8, 5]
[7, 6, 1, 2, 3, 4, 9, 8, 5]
```

MiniZinc found 140 solutions for which the first and last three are shown above. This model employs both arrays and sets. Instead of testing if a particular sum pair is prime, a set of predefined primes {2, 3, 5, 7, 11, 13, 17, 19} serves as a lookup table

in the constraint to efficiently find prime sum pairs: https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/matt_p_card_kennel_puzzle.mzn

## Modeling with Comprehensions

List comprehensions originated from the mathematical set-builder notation, they should be familiar to all who have who have programmed in Python. Aside from Python, functional programming languages like Haskell also support list comprehensions. For those who are unfamiliar or might be a bit hazy on set-builder notation a simple example is given below:

$$S = \{x | x \ \ is \ \ an \ \ int, \ x \geq 0\}$$

This of course describes the integer valued set $\{0,1,2,3,..\}$. MiniZinc's syntax allows for the creation of both list and set comprehensions. Before turning to our first MiniZinc example, we should note that a list or set comprehension is by its nature already a constraint. It should then come as no surprise that comprehensions are a handy construct for the MiniZinc modeler .

For our first example of comprehensions, we consider problem 18 from the 2018 American Mathematics Competition 8 Examination: https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%208/amc8_2018_18.mzn

```
How many positive factors does 23,232 have?
(A)9    (B) 12      (C) 28      (D) 36      (E) 42


% solution set
set of int: ans;

% set comprehension
ans = {i | i in 1..23232 where 23232 mod i = 0};

solve satisfy;

% display total number and solution set
output["Total = "++"\(card(ans))"++" "++"\(ans)"];

Total = 42

{1,2,3,4,6,8,11,12,16,22,24,32,33,44,48,64,66,88,96,
121,132,176,192,242,264,352,363,484,528,704,726,968,1056,
1452,1936,2112,2904,3872,5808,7744,11616,23232}
```

This model employs a set comprehension to specify the solution set. MiniZinc found a total of 42 factors, which are displayed as a set above using the set “card()” function.

A similar set constraint example comes from problem 18 from the 2017 American Mathematics Competition 8 Examination: https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%208/amc8_2017_7.mzn

```
Let Z be a 6-digit positive integer, such as 247247, whose first three
digits are the same as its last three digits taken in the same order.
Which of the following numbers must also be a factor of Z?

A 11   B 19   C 101   D 111   E 1111

% set of possible Z
set of int: nums;

% first three digits are the same as last three digits
nums = {100000*i+10000*j+1000*k+100*i+10*j+k | i in 1..9, j in 0..9, k in
0..9};

% decision variable
var int:d;

% d limited to the 5 options above
constraint d in {11,19,101,111,1111};

% factor requirement
constraint forall(n in nums)(n mod d = 0);

solve satisfy;

% display solution
output["d = "++" ~++"\(d)"];

d =  11
```

This model again employs a set comprehension to restrict a remainder (“mod”) constraint. Out of the 5 given multiple choice options, miniZinc determined that the value of d = 11. Note the search is narrowed the by limiting the decision variable “d” to the 5 given options by constraining the variable “d” to the set {11, 19, 101, 111, 1111}. Lastly, we note the problems 6-digit positive integer requirement (first three digits are the same as its last three digits) is concisely modeled with three separate

conditions (“i in 1..9, j in 0..9, k in  0..9”) in the set comprehension definition of the set “num”.

Another set comprehension problem is 13 from the 2015 American Mathematics Competition 8 Examination:https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%208/amc8_2015_13.mzn

```
How many subsets of two elements can be removed from the set
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11} so that the mean (average) of the
remaining numbers is 6?

(A) 1   (B)  2    (C)    3    (D)    5    (E)    6

%Parent set
set of int: possible;
possible = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

%All possible subsets of the parent set
var set of possible:c;

%Require the cardinality of 9
constraint card(c) = 9;

%average of the remaining numbers is 6
%after a two element subset is removed
constraint sum([n|n in c]) = 54;

solve satisfy;

%Display all two element subsets
output["\(possible diff c)"];

%solutions:     5

{5,7}
{4,8}
{3,9}
{2,10}
{1,11}
```

This model starts off by defining the 11 element integer set “possible”. The decision variable “c” of all possible subsets is constrained by first requiring that “c” have cardinality of 9 (two elements were removed), and secondly that the average of the remaining elements in “c” equals 54 (6*9 = 54) . This accomplished by applying the set function “sum” to the set comprehension “[n|n in c]”. The 5

possible two element subsets are obtained by forming the set difference ("diff") between the parent set "possible" and the set "c".

The generators of a list comprehension usually do not involve decision variables. Nonetheless, as demonstrated below list comprehensions can be used to flatten higher dimensional arrays in constraints.

In combinatorics, a Latin square is defined as an $n \times n$ array filled with $n$ different symbols, each occurring exactly once in each row and exactly once in each column. For example the table below is a 3 x 3 Latin square consisting of the digits $\{1,2,3\}$.

| 1 | 2 | 3 |
|---|---|---|
| 2 | 3 | 1 |
| 3 | 1 | 2 |

The following MiniZinc code uses list comprehensions to generate a Latin square of a given size:

```
% Latin Square
include "alldifferent.mzn";

% size of Latin square to be created
int: N = 5;

% 2-D array decision variable
array[1..N,1..N] of var 1..N:sq;

% row condition
constraint forall(i in 1..N)(alldifferent([sq[i,j]|j in 1..N]));

% column condition
constraint forall(j in 1..N)(alldifferent([sq[i,j]|i in 1..N]));

solve satisfy;

% display magic square
output[show(sq)];
```

```
[3, 1, 2, 4, 5,
 1, 2, 5, 3, 4,
 2, 5, 4, 1, 3,
 4, 3, 1, 5, 2,
 5, 4, 3, 2, 1]
```

Since the global constraint “alldifferent” operates only on one-dimensional arrays, a list comprehension is used in both the row and column constraints to flatten the “sq” array into “N” one-dimensional arrays. One of the many 5 x 5 Latin squares generated by MiniZinc is listed after the code.

In recreational mathematics, a magic square is a grid of numbers whose row, column, and diagonal sums are all equal. This common value is usually referred to as the magic constant. If the square is to be filled with the integers $\{1,2,3,...,n^2\}$ then the magic constant can be seen to equal $\frac{n(n^2+1)}{2}$. Consider the model below, which generates a 5 x 5 magic square: https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/mgic_sq.mzn

```
% Magic Square

include "alldifferent.mzn";
% size of magic square to be created
int: N = 5;

% 2-D array decision variable
array[1..N,1..N] of var 1..N*N:sq;

% magic constant
int:magic_num = (N*(N*N+1)) div 2;

% list comprehension
constraint alldifferent([sq[i,j]|i in 1..N ,j in 1..N]);

% row condition
constraint forall(i in 1..N)(sum(j in 1..N)(sq[i,j]) = magic_num);

% column condition
constraint forall(j in 1..N)(sum(i in 1..N)(sq[i,j]) = magic_num);

% diagonal conditions
constraint forall(i in 1..N)(sum(i in 1..N)(sq[i,i]) = magic_num);
constraint forall(i in 1..N)(sum(i in 1..N)(sq[N+1-i,i]) = magic_num);

solve satisfy;

% display magic square
```

```
output[show(sq)];

  [25, 5,  9,   3, 23,
   4,  24, 15, 14,  8,
   20, 7,   1, 16, 21,
   6,  17, 18, 13, 11,
   10, 12, 22, 19, 2]
```

Once again since the “alldifferent” global constraint operates only on one-dimensional arrays, a list comprehension is used to flatten the “sq” array into a one-dimensional array. In contrast, the row, column, and diagonal constraints are expressed in terms of the original two-dimensional array. The 5 x 5 magic square with a magic sum of 65 is listed after the code.

We would be amiss if at this point we failed to solve a Sudoku puzzle with MiniZinc code. It should be noted that a Sudoku puzzle is a special case of a Latin square. Any solution to a Sudoku puzzle is also Latin square. Sudoku puzzles have the additional constraint that in the case of a 9 x 9 grid (the most common Sudoku configuration) all 3 × 3 adjacent sub-squares must contain each of the digits $\{1,2,3,...,9\}$ exactly once. Before turning to a MiniZinc program to solve a Sudoku puzzle, a completed 4 x 4 is given below:

| | | | |
|---|---|---|---|
| 2 | 4 | 1 | 3 |
| 1 | 3 | 2 | 4 |
| 3 | 1 | 4 | 2 |
| 4 | 2 | 3 | 1 |

We now turn to the MiniZinc Sudoku code: