

Discrete Optimisation Exercise: Comprehensions and Generators

Introduction

In this exercise, we will focus on **comprehensions** (and the related concept of **generator calls**). Comprehensions are important in MiniZinc because they are the only way to build up arrays.

Part 1 - Simple comprehensions

A basic comprehension has the syntax `[e | i in g]`. Here, `i` is an identifier, `g` is a set or an array (which we call the *generator*), and `e` is an expression (which typically contains `i`).

You can think of a comprehension as a loop: The variable `i` is iterated over the set or array `g`, and in each iteration, the element `e` is generated.

Step 1 Write a MiniZinc model that creates an array of the first ten multiples of 3 (starting with 3). Start from the following and replace the `TODO` comment with your own code.

```
array[int] of int: multiples_of_three ::output = [ /* TODO: insert answer */ ];
```

The output should be:

```
multiples_of_three = [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]  
----
```

Step 2 We can use **where** clauses to restrict the elements in a comprehension. A **where** clause is tested for each iteration, and the element `e` is only generated if the where clause is true for that iteration.

Add the following to your model to compute only the even multiples of 3:

```
array[int] of int: even_multiples_of_three ::output = [ /* TODO: insert answer */ ];
```

The resulting output should be

```
even_multiples_of_three = [6, 12, 18, 24, 30];  
-----
```

Step 3 Comprehensions can use multiple iterators and generators: `[e | i in g1, j in g2]` iterates over both `g1` and `g2`. Note that the second iterator is the “faster” one: for each value of `i`, the comprehension will loop through all values of `j` before incrementing `i`. If `g1=g2`, you can also write `[e | i, j in g2]`.

Write code that creates an array of all products of pairs of numbers between 1 and 10:

```
array[int] of int: products ::output = [ /* TODO: insert answer */ ];
```

The resulting output should be

```
products = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
2, 4, 6, 8, 10, 12, 14, 16, 18, 20,
3, 6, 9, 12, 15, 18, 21, 24, 27, 30,
4, 8, 12, 16, 20, 24, 28, 32, 36, 40,
5, 10, 15, 20, 25, 30, 35, 40, 45, 50,
6, 12, 18, 24, 30, 36, 42, 48, 54, 60,
7, 14, 21, 28, 35, 42, 49, 56, 63, 70,
8, 16, 24, 32, 40, 48, 56, 64, 72, 80,
9, 18, 27, 36, 45, 54, 63, 72, 81, 90,
10, 20, 30, 40, 50, 60, 70, 80, 90, 100];
-----
```

Now extend this model to only produce the products that are divisible by 3 but not divisible by 2, without any duplicates:

```
array[int] of int: products_div_3_not_2 :: output = [ /* TODO: insert answer */ ];
```

The output should be

```
products_div_3_not_2 = [3, 9, 15, 21, 27, 45, 63, 81];
-----
```

Hint: you may want to use a *set comprehension* rather than array comprehension, since this will automatically remove duplicates. If you assign a set to an array, MiniZinc will automatically coerce the set to a sorted array of its elements.

Part 2 - Multi-dimensional arrays

Multi-dimensional arrays are useful to group related things together. For example, you could have a two-dimensional map, and for each pair of x and y coordinates, you may want to store the reward for visiting that square of the map.

The comprehensions we have seen so far are always one-dimensional. In order to create a multi-dimensional array, we have to do one of two things.

The traditional way to create multidimensional arrays is to coerce the result of a 1D comprehension into the correct dimensions. We use the `arrayXd` family of functions for this, where `X` stands for the number of dimensions. For example, the following would turn our `products` array from before into a two-dimensional array:

```
array[int,int] of int: products_2d = array2d(1..10, 1..10, products);
```

Note: We have to specify the index sets for each dimension.

Alternatively, since MiniZinc 2.6 we can provide the indexes of each element in the comprehension using the syntax `[(e2,e3,...): e1 | i in g1, j in g2, ...]`. Here, `e2`, `e3`, etc. are expressions used to calculate the index of the current iteration in the resulting array. For example the two-dimensional `products` array could also be defined as:

```
array[int,int] of int: products_2d = [(i,j): i * j | i,j in 1..10];
```

Step 1 Given the x and y coordinates for certain points:

```
set of int: Point = 1..5;
array[Point] of int: x_coord = [1,3,5,1,10];
array[Point] of int: y_coord = [4,3,8,1,7];
```

Create an array that contains the Manhattan distance between each pair of points:

```
array[Point,Point] of int: manhattan_distance ::output = [ /* TODO: insert answer */ ];
```

The output should be

```
[| 0, 3, 8, 3, 12
| 3, 0, 7, 4, 11
| 8, 7, 0, 11, 6
| 3, 4, 11, 0, 15
| 12, 11, 6, 15, 0
|];
```

Step 2 Let's assume that we want to add a “dummy point”, with a distance of 0 to all real points. Write a comprehension that creates the distance matrix by using the previously computed `manhattan_distance` if both points are in the original set `Points`, and 0 otherwise:

```
set of int: Point0 = 0..5;
```

The resulting output should contain an extra row and an extra column of zeroes:

```
[| 0: 1: 2: 3: 4: 5:
| 0: 0, 0, 0, 0, 0, 0
| 1: 0, 0, 3, 8, 3, 12
| 2: 0, 3, 0, 7, 4, 11
| 3: 0, 8, 7, 0, 11, 6
| 4: 0, 3, 4, 11, 0, 15
| 5: 0, 12, 11, 6, 15, 0
|];
```

Notice how, when the index sets are not `1..n` then MiniZinc makes the index sets explicit.

Step 3 Repeat Step 3 but with Euclidean distance instead of Manhattan distance. The resulting array should contain `float` numbers

```
[| 0: 1: 2: 3: 4: 5:
| 0: 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
| 1: 0.0, 0.0, 2.23606797749979, 5.656854249492381, 3.0, 9.486832980505138
| 2: 0.0, 2.23606797749979, 0.0, 5.385164807134504, 2.82842712474619, 8.062257748298549
| 3: 0.0, 5.656854249492381, 5.385164807134504, 0.0, 8.062257748298549, 5.099019513592784
| 4: 0.0, 3.0, 2.82842712474619, 8.062257748298549, 0.0, 10.81665382639197
| 5: 0.0, 9.486832980505138, 8.062257748298549, 5.099019513592784, 10.81665382639197, 0.0
|];
```

Part 3 - Generator Calls

A generator call is a function call of the form `f (i in g) (e)`. You have seen these many times already, for example with the `forall` or `sum` functions. In MiniZinc, a generator call is really just

a shorthand notation for writing a comprehension:

```
constraint forall (i in 1..n-1) (x[i] < x[i+1]);
```

is exactly the same as writing

```
constraint forall ( [ x[i] < x[i+1] | i in 1..n-1 ] );
```

So, if you know how to write comprehensions, you also know how to write generator calls! Let's now assume we have an array of n integer variables:

```
int: n;  
array[1..n] of var 1..n: x;
```

Now we want to constrain x to be pairwise different, but let's say we don't have the `all_different` constraint in the library, so we'll have to write it ourselves. The basic idea is to write the following:

"All pairs of variables x_i and x_j in x cannot be equal."

The tricky part is to define what we mean by "pairs of variables". Clearly, we don't mean e.g. $x[1] \neq x[1]$, because that would be unsatisfiable. So we can't just write

```
constraint forall (xi, xj in x) (xi != xj); %% THIS IS WRONG
```

You may think the following could work:

```
constraint forall (xi, xj in x where xi != xj)  %% THIS IS ALSO WRONG  
    (xi != xj);
```

Why is this also wrong? Because the where clause is meaningless: It talks about the value of x_i being different from x_j . We need to make sure that the indexes are different.

Step 1 There are multiple ways (at least 3) to express that all pairs of variables in x must take different values. Write down all that you can come up with.

Step 2 In all of our code above, we assumed that we can write down the sets to iterate over. E.g., we iterated over `Point`, or `1..n`. MiniZinc provides functions to query the index set of an array expression. That way, we can write comprehensions that are more general. The functions are `index_set(x)` (if x is a one-dimensional array), and `index_set_1of2(x)` and `index_set_2of2(x)` for two-dimensional arrays (similar functions exist for 3 and more dimensions).

Rewrite your code to use `index_set` instead of concrete sets.

Part 4 - Output

Comprehensions are also very useful for producing readable output. The output statement in MiniZinc expects an array of strings, which you can of course construct using a comprehension.

In order to turn any variable or parameter into a string for output, you can use the syntax `"some string \(\mathbf{e}\) some more string"`. The expression \mathbf{e} is converted to a string and inserted into the surrounding string.

Note: The output for step `n` should be defined using the `output_stepn` variable declaration. You can uncomment the `output output_stepn;` line to see how your output looks.

Step 1 For the `even_multiples_of_three`, create output that looks like this:

```
Even multiple number 1 is 6
Even multiple number 2 is 12
Even multiple number 3 is 18
Even multiple number 4 is 24
Even multiple number 5 is 30
-----
```

You should use the `index.set` function for this step.

Step 2 For the `manhattan_distance`, create output that looks like this:

```
Distance between 1 and 2 is 3
Distance between 1 and 3 is 8
Distance between 1 and 4 is 3
Distance between 1 and 5 is 12
Distance between 2 and 3 is 7
Distance between 2 and 4 is 4
Distance between 2 and 5 is 11
Distance between 3 and 4 is 11
Distance between 3 and 5 is 6
Distance between 4 and 5 is 15
-----
```

Instructions

Edit the provided `mzn` model files to solve the problems described above. The MiniZinc IDE and the online auto-grader will give you feedback on your solution.