FIT5216: Modelling Discrete Optimization Problems
Flattening Lab Exercise

1 Introduction

The aim of this lab exercise is to determine the result of flattening a number of small MiniZinc models. Normally, the result of flattening is saved into a temporary file and discarded after solving, but you can save or output this FlatZinc file by using the command-line flags `--fzn <file>`, `--output-fzn-to-file <file>`, `--output-to-stdout` or `--output-fzn-to-stdout` (e.g., `minizinc model.mzn data.dzn --fzn model.fzn`). You can do the same in the IDE using the dropdown MiniZinc menu item Compile, which creates and loads the resulting FlatZinc file. It can also be run, using the Run button.

1.1 The MiniZinc Model - `flat_intro.mzn`

The following MiniZinc model `flat_intro.mzn`:

```
var -2..3: x;
var 0..5: y;
constraint x * (3*y + x - 4) + 4 * y <= 4;
```

flattens to a FlatZinc model (`flat_intro.fzn`)[1]:

```
array [1..2] of int: X_INTRODUCED_0_ = [1,4];
var -2..3: x:: output_var;
var 0..5: y:: output_var;
var -6..14: X_INTRODUCED_2_ ::var_is_introduced :: is_defined_var;
var -28..42: X_INTRODUCED_3_ ::var_is_introduced :: is_defined_var;
constraint int_lin_le(X_INTRODUCED_0_,[X_INTRODUCED_3_,y],4);
constraint int_lin_eq([1,3,-1],[x,y,X_INTRODUCED_2_],4):: defines_var(X_INTRODUCED_2_);
constraint int_times(x,X_INTRODUCED_2_,X_INTRODUCED_3_):: defines_var(X_INTRODUCED_3_);
solve  satisfy;
```

Rewriting this back into a more concise MiniZinc model:

```
var -2..3: x;
var 0..5: y;
var -6..14: X_2 = x + 3*y - 4;
var -28..42: X_3 = x * X_2;
constraint X_3 + 4*y <= 4;
```

[1]Note that in this newer version of MiniZinc (2.5.5), the naming of introduced variables has changed from e.g. `INT01` (as shown in the lectures) to `X_INTRODUCED_i_`

1.2 Flattening Expressions - `flat_exp.mzn`

Write down the MiniZinc form of the flat model that results from the following `flat_exp.mzn`

```
var -2..3: x;
var 0..5: y;
var -1..2: u;
var 0..3: v;
constraint (x - u)*(x - u) + (y - v)*(y - v) >= 15;
constraint (x- u)*(y - v) >= 0;
```

Verify that they give the same results. To do this, we need to output all the solutions. However, the rewrite might change the order of the solutions, which will make them hard to compare. The most easy fix for this is to add a fixed search strategy to the `solve` statement which will ensure that all solutions are produced in the same order:

```
solve :: int_search([x,y,u,v], input_order, indomain_min) satisfy;
```

Look at the generated FlatZinc only after you have generated your answer.

1.3 Unrolling - `flat_unroll.mzn`

Write down the MiniZinc form of the flat model that results from the following `flat_unroll.mzn`

```
array[1..4] of int: a = [1,4,6,3];
array[1..4] of int: b = [3,2,1,4];
array[1..4] of var 0..9: x;
constraint forall(i in 1..4 where a[i] < b[i])(x[i] >= 4);
constraint sum(i in 2..4)((a[i] - b[i]) * x[i]) >= 37;
constraint forall(i in 1..4)(x[1] <= x[i]);
solve minimize sum(x);
```

Verify that they give the same results. Again, you can do this most simply by giving them a fixed search strategy so that the solutions are produced in the same order. Additionally, you will have to replace the `solve` item to `solve satisfy`), otherwise the solutions will be the *intermediate* solutions instead of all the solutions. Look at the generated FlatZinc only after you have generated your answer.

1.4 Flattening Arrays - `flat_2d.mzn`

Write down the MiniZinc form of the flat model that results from the following `flat_2d.mzn`

```
array[-1..1,-2..1] of var -1..1:x;
constraint forall(i in -1..1)(x[i,i] >= 1);
constraint sum(i in -2..1)(x[0,i]) = 0;
constraint x[x[1,1],x[0,0]] = 1;
constraint forall(i in -1..1)(x[i,-2] = 0);
```

Remember that the 2D array must be one dimensional. You might want to go in two steps, first flatten, then translate to 1D. Verify that they give the same results. You can do this most simply by giving them a fixed search strategy and checking that all solutions are produced in the same order. Look at the generated FlatZinc only after you have generated your answer.

1.5 Flattening Functions - `flat_func.mzn`

Write down the MiniZinc form of the flat model that results from the following `flat_func.mzn`

```
function var int: myabs(var int: x) =
                  let { var int: y;
                        constraint y >= 0;
                        constraint y = x \/ y = -x; } in
                  y;

array[1..3] of var -2..2: x;
constraint sum(i in 1..3)(myabs(x[i])) <= 2;
```

Verify that they give the same results. You can do this most simply by giving them a fixed search strategy and checking that all solutions are produced in the same order. Look at the generated FlatZinc only after you have generated your answer.

1.6 Flattening Torture - `flat_torture.mzn`

Write down the MiniZinc equivalent of the FlatZinc resulting from

```
array[1..3] of var 0..4: x;
constraint exists(i in 1..5)
                 ( let { var 0..3: y = x[i] - 1; } in
                   y <= 0 );
constraint (let { var int: y = x[x[1]] } in y * y > 0) -> x[1] = 0;
```

Verify that they give the same results. You can do this most simply by giving them a fixed search strategy and checking that all solutions are produced in the same order, i.e. Look at the generated FlatZinc only after you have generated your answer.