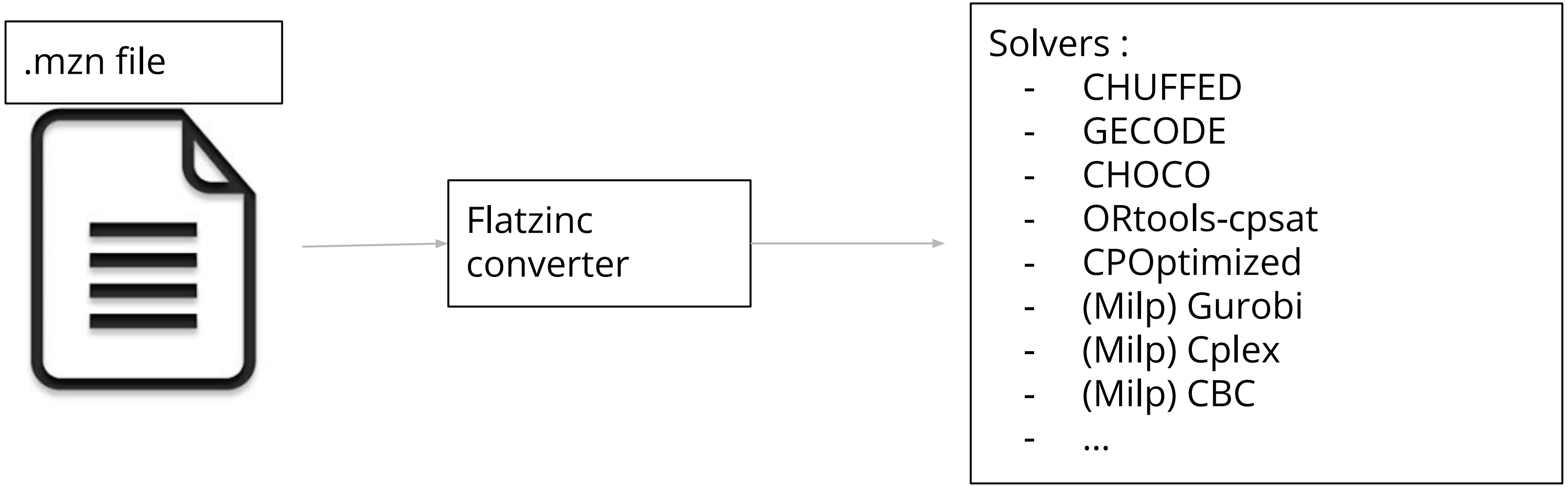
Minizinc introduction

A brief introduction to the constraint programming/decision making language

MINIZINC

Minizinc is a declarative language designed to specify constrained mathematical problems in an extensive way. It is not solver oriented but modelling oriented. However some instructions can be given to the solver directly in minizinc annotations.

Minizinc is designed to be interfaced with many solver backend, that are using flatzinc format.



Solvers :

- CHUFFED
- GECODE
- CHOCO
- ORtools-cpsat
- CPOptimized
- (Milp) Gurobi
- (Milp) Cplex
- (Milp) CBC
- ...

First motivation example : simple addition

```
% Input
int: x=2;
int: y=2;

% Decision variable
var int: z;
% Addition constraint
constraint z=x+y;

% Solve command
solve satisfy;

% Output
output ["z=x+y : \(z)=\(x)+\(y)"];
```

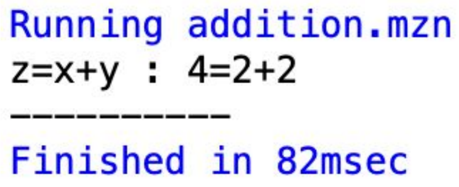
(Constant) input of the problem

Variable integer

Constraint for the simple addition

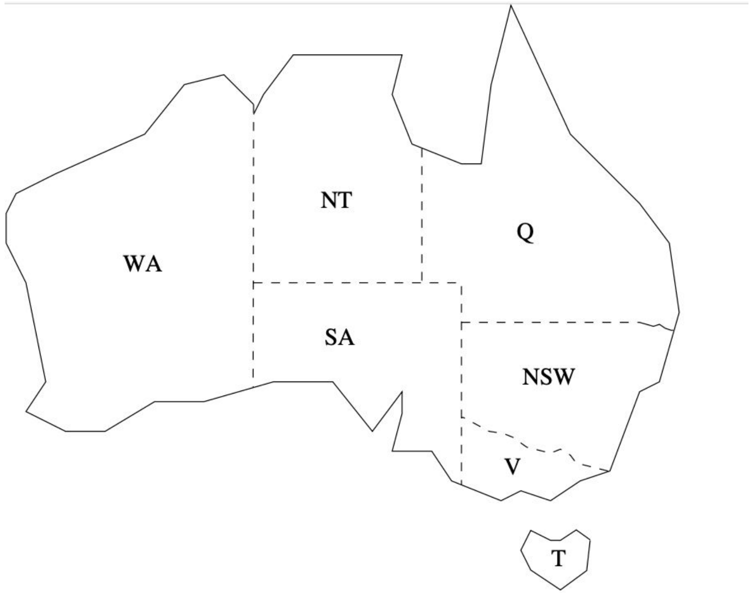
Solve command

Custom Output



```
Running addition.mzn
z=x+y : 4=2+2
----------
Finished in 82msec
```

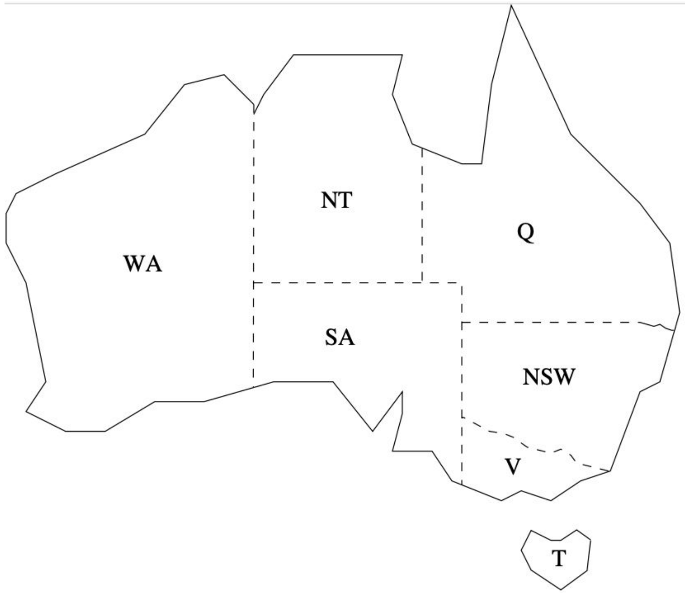
Classical example : coloring problem



The goal is to color the map so that adjacent region are not with the same color. It is the well known coloring problem. On planar maps it is proven that you can color with 4 colors maximum. (ponctual intersection with more than 3 areas is not considered as a common border)

https://www.minizinc.org/doc-2.5.5/en/modelling.html

Classical example : coloring problem

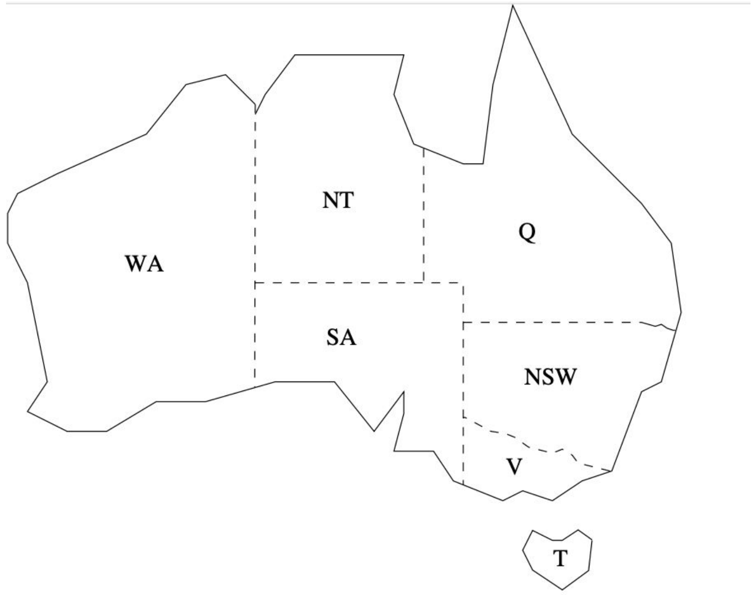


Formulation :
For all region r in [WA, NT, Q, SA, NSW, V, T]
Color[r] is some integer value.

Color vector has some constraint. Based on the input map of australia.

https://www.minizinc.org/doc-2.5.5/en/modelling.html

Classical example : coloring problem : correction



```
% Colouring Australia using nc colors
int: nc=4;
set of int: range_color=1..nc;

% Variable color for each region.
var range_color: wa;
var range_color: nt;
var range_color: sa;
var range_color: q;
var range_color: nsw;
var range_color: v;
var range_color: t;

constraint wa != nt;
constraint wa != sa;
constraint nt != sa;
constraint nt != q;
constraint sa != q;
constraint sa != nsw;
constraint sa != v;
constraint q != nsw;
constraint nsw != v;

solve satisfy;
```

https://www.minizinc.org/doc-2.5.5/en/modelling.html

Go further

Generic coloring problem. Let’s consider a graph with n_nodes, n_edges. We want a model that works for any input graph data. Minizinc is perfect for that !

.mzn file

```
int: n_nodes;
int: n_edges;
int: nb_colors;
array[EDGES, 1..2] of NODES: list_edges;
set of int: NODES = 1..n_nodes;
set of int: EDGES = 1..n_edges;
set of int: COLORS = 1..nb_colors;
```

```
n_nodes=10;
n_edges=5;
nb_colors=4
list_edges=[|1,2,|3,4,|2,5,|4,6,|7,8|];
```

.dzn file

Model file

Data file

Syntax convention : input and variables

Range of int value :
set of int: range_numbers=1..n;

1D array of int (using indexes range_numbers) :
array[range_numbers] of int: array_numbers = [1 | i in range_numbers];

1D array of variable int :
array[range_numbers] of var int: array_numbers;

Variable set of int:
var set of int: variable_set;

2D array of int :
array[range_numbers, range_numbers] of int: array_number2d=[|1,2,|3,4|];
Equivalent to the use of “array2d” constructor.
array[range_numbers, range_numbers] of int: array_number2d=array2d(range_numbers, range_numbers,[1,2,3,4]);

Syntax convention : Constraints

Minizinc includes a wide range of mathematics literal to express a wide variety of constraints

- “**<,<=,>,>=,=/==,!=**” compators
- “**/\**” : logical and, “\/” : logical or, “not” negation
- “**->**” implies
- “**<->**” equivalence (on if and only if)
- **forall**(i in some_set)(expression) : very useful to specify a constraint in a loop way
- **exists**(i in some_set)(expression) : very useful to specify an “there exists”' constraint.

When you have a doubt, https://www.minizinc.org/doc-2.5.5/en/part_2_tutorial.html will probably be of great help.

Global constraints

Minizinc includes a wide range of global constraints that are of great use to improve the model efficiency in practice. Indeed the implementation of those global constraints in different solver backend can be very efficient in term of propagation of the constraints.

Some good practice:

1) **When you can model a big number of constraints via a global constraint**
2) **When It can reduce the state space size extensively without changing the mathematical problem**
3) **Channeling constraints between redundant variables (https://www.minizinc.org/doc-2.5.5/en/lib-globals.html?highlight=channelling#channeling-constraints)**

Some very used global constraint

-**all_different**(array[int] of var int) : all element of the array should get different value
-**all_different_except_0** : all element of the array should get different value or 0.
-**global_cardinality**(array [$X] of var int: x,array [$Y] of int: cover, array [$Y] of var int: counts)
Requires that the number of occurrences of cover [i] in x is counts [i].
-lexicographic/ordering constraints (value_precede_chain, seq_precede_chain)...
-**inverse**(array [int] of var int: f, array [int] of var int: invf) :
If f[i]==j then invf[j]==i. This can be useful for problems where you can see from **different** angles. Some constraints can be easily written in one or the other angle so it is sometimes useful to handle 2 variables.
-**cumulative(starts, duration, consumption, ressource_capacity) :**

Come back to the coloring problem

Compared to the Australia problem, We want to minimize the number of colors ! and it is not trivial to have an upper bound so the “nb_colors” can be chosen quite high.

```
int: n_nodes;
int: n_edges;
int: nb_colors;
array[EDGES, 1..2] of NODES: list_edges;
set of int: NODES = 1..n_nodes;
set of int: EDGES = 1..n_edges;
set of int: COLORS = 1..nb_colors;
array[NODES] of var COLORS: color_graph;
```

Your take !

Correction

```
int: n_nodes;
int: n_edges;
int: nb_colors;
set of int: NODES = 1..n_nodes;
set of int: EDGES = 1..n_edges;
set of int: COLORS = 1..nb_colors;
array[EDGES, 1..2] of NODES: list_edges;
array[NODES] of set of NODES: graph=[{list_edges[e, 2]| e in EDGES where list_edges[e,1]=n} union
                                    {list_edges[e, 1]| e in EDGES where list_edges[e,2]=n} | n in NODES];

include "alldifferent.mzn";
include "value_precede_chain.mzn";
bool: include_seq_chain_constraint;
constraint if include_seq_chain_constraint then
            value_precede_chain([c | c in COLORS],
                                color_graph)
        endif;
constraint forall(n in NODES)(forall(n_n in graph[n] where n_n>n)(color_graph[n]!=color_graph[n_n]));
array[NODES] of var COLORS: color_graph;
var int: obj=max(color_graph);
solve ::int_search(color_graph, smallest, indomain_min, complete) minimize(obj);
output ["obj=\(obj)"];
```

References

Basic modeling course,
https://www.coursera.org/learn/basic-modeling/home/welcome

Minizinc website, doc. https://www.minizinc.org/,
https://www.minizinc.org/doc-2.5.5/en/index.html,
https://www.minizinc.org/doc-2.2.1/en/MiniZinc%20Handbook.pdf

Minizinc examples https://github.com/MiniZinc/minizinc-examples,
https://github.com/MiniZinc/minizinc-benchmarks