# Basic Modelling

MONASH
University

# Modelling Discrete Optimisation Problems

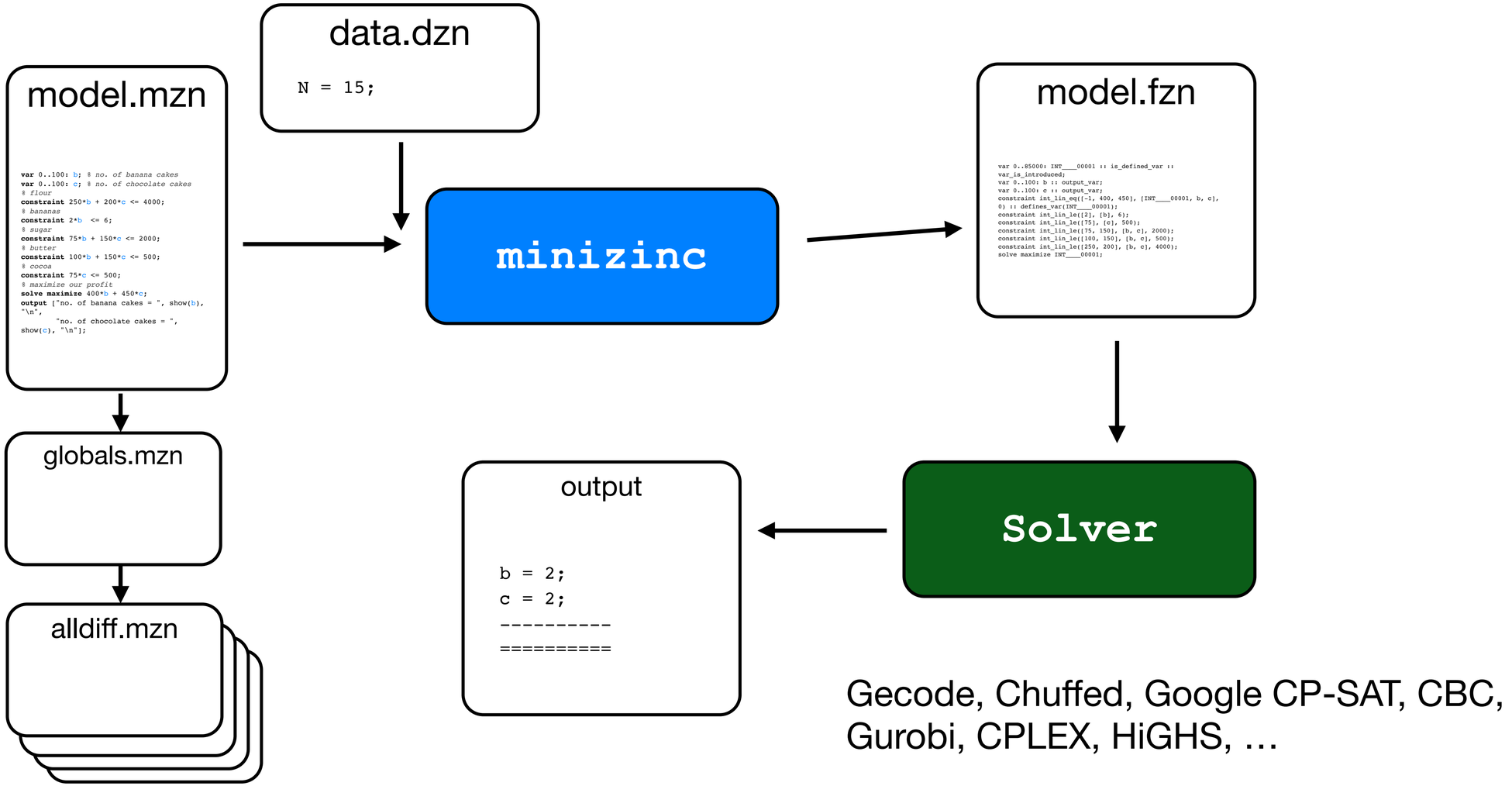
What kind of problems are we talking about?

- NP-hard satisfaction and optimisation problems
- A finite set of decision variables, usually with finite domains (e.g. bool, int, set of int, enumerated types)
- A finite set of constraints, restricting the combinations of values the variables can take
- (Optionally) An objective function to minimise or maximise

# The MiniZinc Mantra

- Modelling and solving Discrete Optimization problems is **hard**
- There are **many ways** to solve these problems
- We should
  - model them
    - **once**, and
    - rapidly
  - **solve** them
    - with many different technologies
    - in many different ways
    - rapidly

# MiniZinc Architecture




Gecode, Chuffed, Google CP-SAT, CBC, Gurobi, CPLEX, HiGHS, …

# Software Distribution

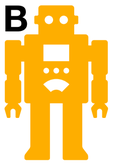
- Download from https://www.minizinc.org
  - MiniZinc compiler, IDE, default solvers (Gecode, Chuffed, COIN-BC, HiGHS, Google CP-SAT, and interfaces for Gurobi & CPLEX)
- Or use online browser-based version at https://play.minizinc.dev

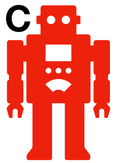
# First Steps

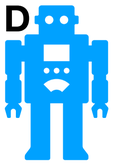
Peter Stuckey & Guido Tack

MONASH
University

# Setting Up the Toy Shop


| | | | | |
|---|---|---|---|---|
| Toys per hour | 6 | 10 | 8 | 40 |
| Cost per robot | $13k | $21k | $17k | $100k |
| Robots available | 1000 | 400 | 500 | 150 |

Choose how many of each type of robot to buy, within a budget.
Goal: maximise total number of toys per hour produced.

# Decisions, Constraints and Objectives

- **Decisions** are the unknowns of a problem:
  - how many of each robot to buy?
- **Constraints** place restrictions on the decisions:
  - limit on availability of the robots
  - total budget for buying robots: $10M
- The **Objective** specifies our goal:
  - maximising total production per hour

# Simple Strategy 1: Most productive robots

- Buy as many of the most expensive robots as possible
  - 100 robots of type D
  - total production: 4000/hour


| | | | | |
|---|---|---|---|---|
| Toys per hour | 6 | 10 | 8 | 40 |
| Cost per robot | $13k | $21k | $17k | $100k |
| Robots available | 1000 | 400 | 500 | 150 |

# Simple Strategy 2: Cheapest Robots

- Buy as many robots as possible given the budget
  - 769 robots of type A
  - total production: 4614/hour
- **Can we do better?**


| | A | B | C | D |
|---|---|---|---|---|
| Toys per hour | 6 | 10 | 8 | 40 |
| Cost per robot | $13k | $21k | $17k | $100k |
| Robots available | 1000 | 400 | 500 | 150 |

# Mathematical model

$$maximize\ 6A + 10B + 8C + 40D$$

linear objective

$$subject\ to$$

$$13A + 21B + 17C + 100D \leq budget$$

linear constraint

$$0 \leq A \leq 1000$$
$$0 \leq B \leq 400$$
$$0 \leq C \leq 500$$
$$0 \leq D \leq 150$$

$$A, B, C, D \in \mathbb{Z}$$
$$budget = 10000$$

# Toy Shop MiniZinc model

```
int: budget = 10000;

var 0..1000: A;
var 0..400:  B;
var 0..500:  C;
var 0..150:  D;


constraint 13*A + 21*B + 17*C + 100*D <= budget;

solve maximize 6*A + 10*B + 8*C + 40*D;
```

parameter definitions

decision variable declarations

constraint(s)

objective

# Parameters

- MiniZinc has two kinds of variables
- The first kind is **parameters**
  - These are like variables in a standard programming language. They must be assigned a value (but only **once**).
  - They are declared with a type: `int`, `float`, or `bool` (or a range/set)
  - The following are logically equivalent
    - `int: i=3;`
    - `int: i;  i=3;`

# Decision Variables

- The other kind is **decision variables**:
  - like variables in mathematics
  - declared with `var` and a type (or a range/set)
  - can also be *assigned* once with a fixed value expression
- **Range**: written `l..u`
  - a contiguous sequence of integers from `l` to `u`
- The following are logically equivalent:
  - `var int: i; constraint i >= 0; constraint i <= 4;`
  - `var 0..4: i;`
  - `var {0,1,2,3,4}: i;`
- The following are also logically equivalent
  - `var int: i = x + 3;`
  - `var int: i; constraint i = x + 3;`

# Constraints

- Basic arithmetic constraints are built using the standard arithmetic relational operators:

  `=` `!=` `>` `<` `>=` `<=`

- You can also use Unicode versions: `≠` `≤` `≥`

- Constraints in MiniZinc are written in the form

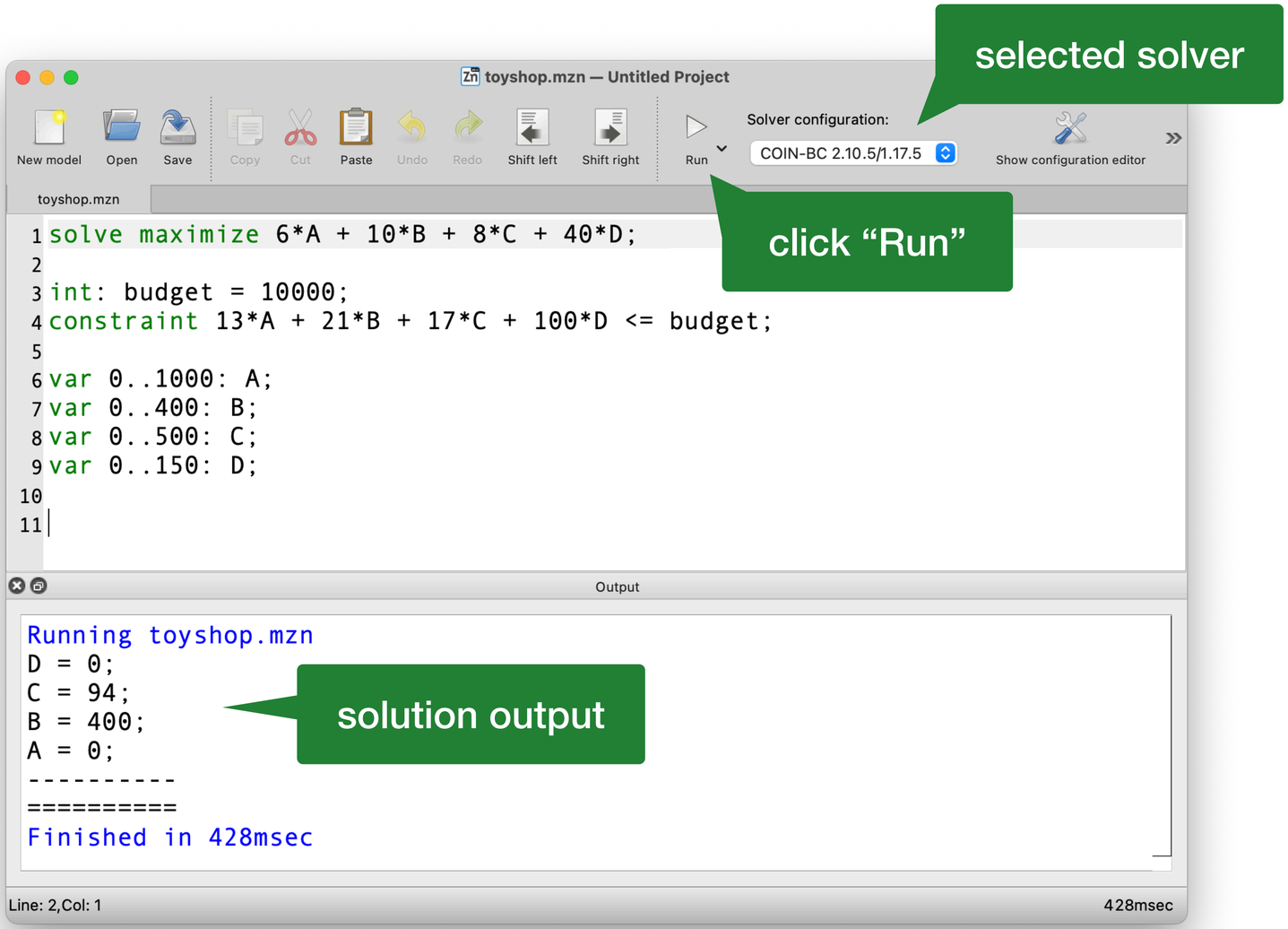
  `constraint` *`<constraint-expression>`*

# Objective

- Specifies what we want to achieve:
  - `solve maximize <expr>;`
  - `solve minimize <expr>;`
  - `solve satisfy;`

this is optional

# Running the Model



# Running the Model

```
 9 var 0..150: D;
10
11
```

Output

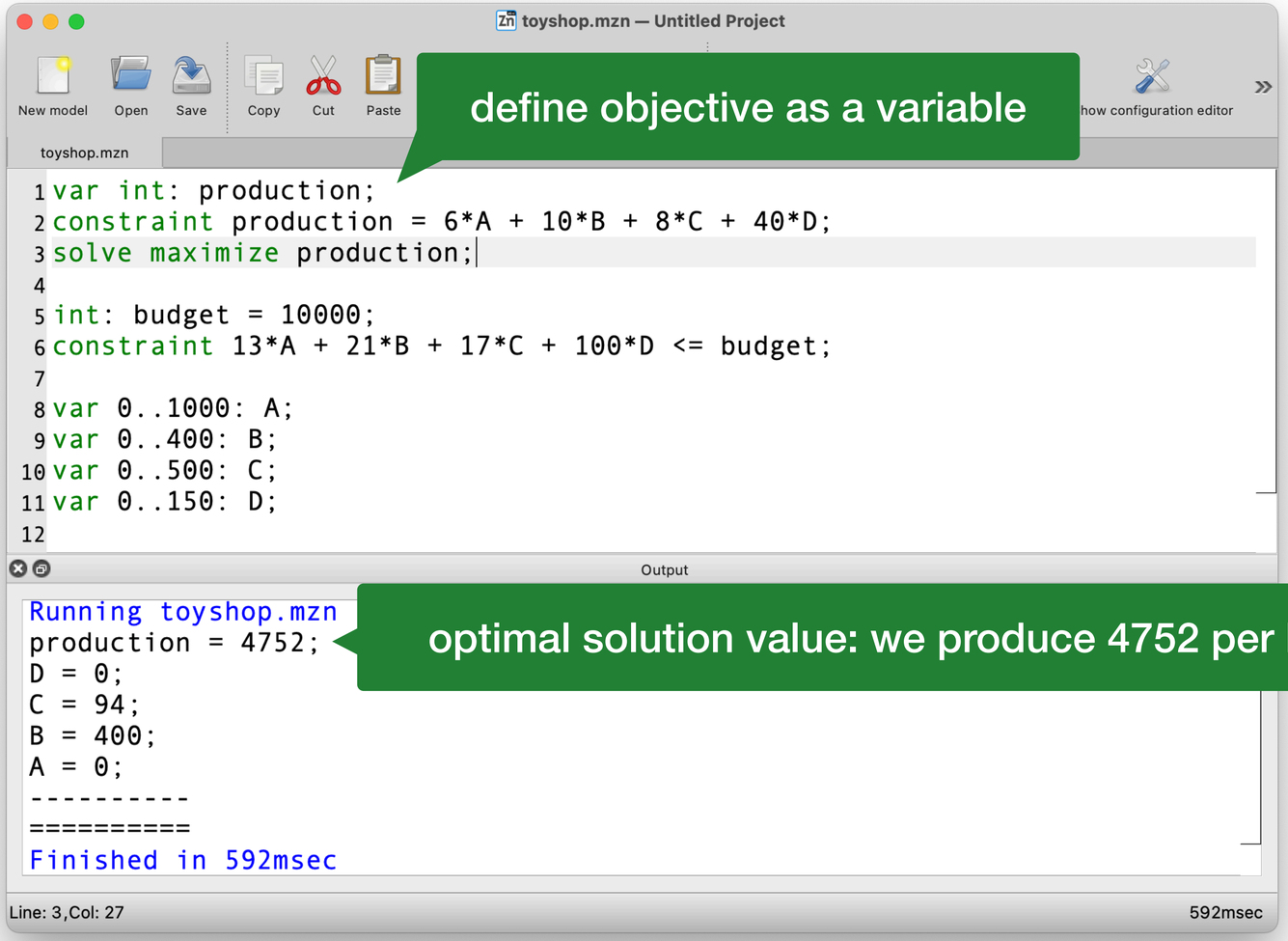
```
Running toyshop.mzn
D = 0;
C = 94;
B = 400;
A = 0;
----------
==========
Finished in 428msec
```

Line: 2,Col: 1

ten dashes indicates solution

ten equals signs indicates no better solution (i.e., the last solution was optimal)

# Running the Model




define objective as a variable

optimal solution value: we produce 4752 per hour

# Command line use *(optional)*

- You can also call MiniZinc from the command line:

```
$ minizinc --solver cbc toyshop.mzn
```

- The output is printed in the terminal:

```
production = 4752;
D = 0;
C = 94;
B = 400;
A = 0;
----------
==========
```

# Summary

- We can directly describe/write and solve discrete optimisation problems in MiniZinc.
- **Linear** models are the most common form of modelling
  - **linear programming** has been used since the 1940s
  - **integer linear** constraints are hard to solve (NP-hard)
  - critical part of almost any discrete optimisation problem!

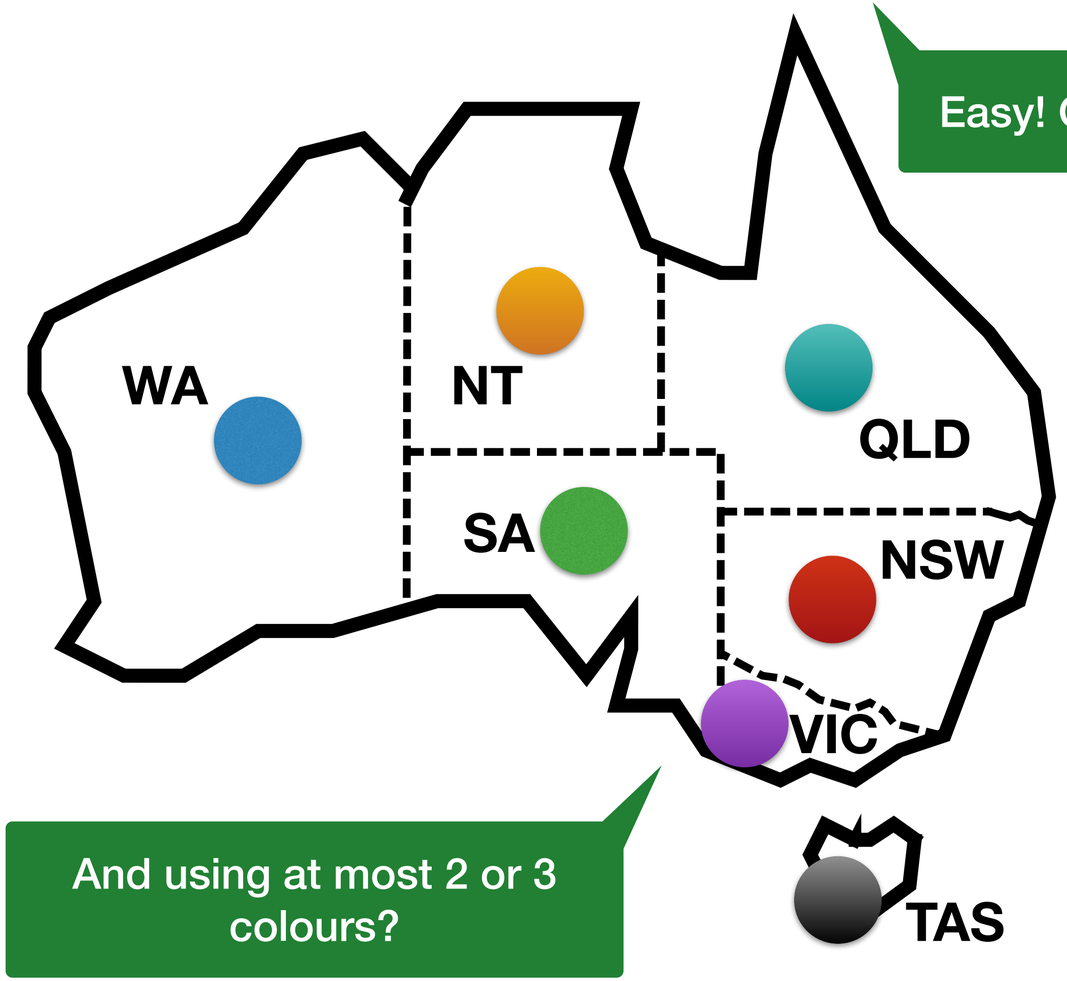
# Map colouring

Peter Stuckey & Guido Tack

MONASH
University

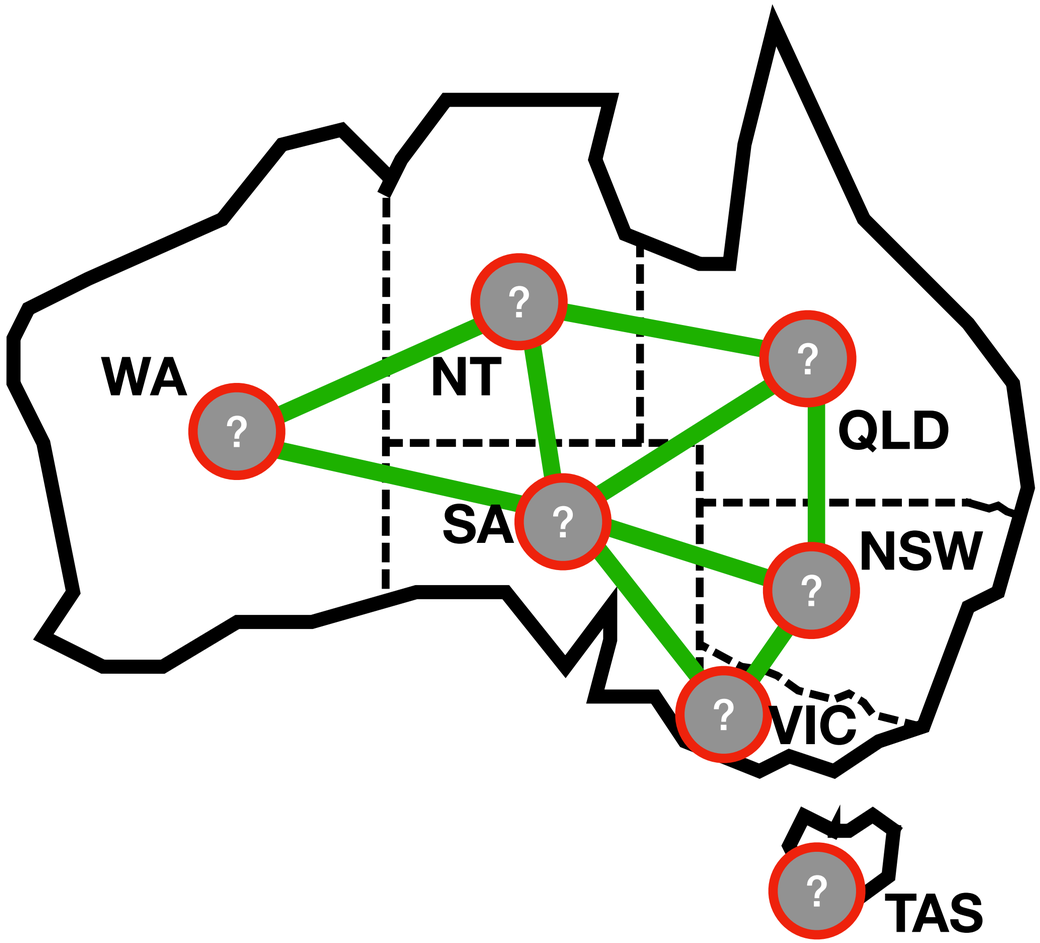
# Map Colouring

Goal: colour adjacent states with different colours



# Map Colouring is Graph Colouring

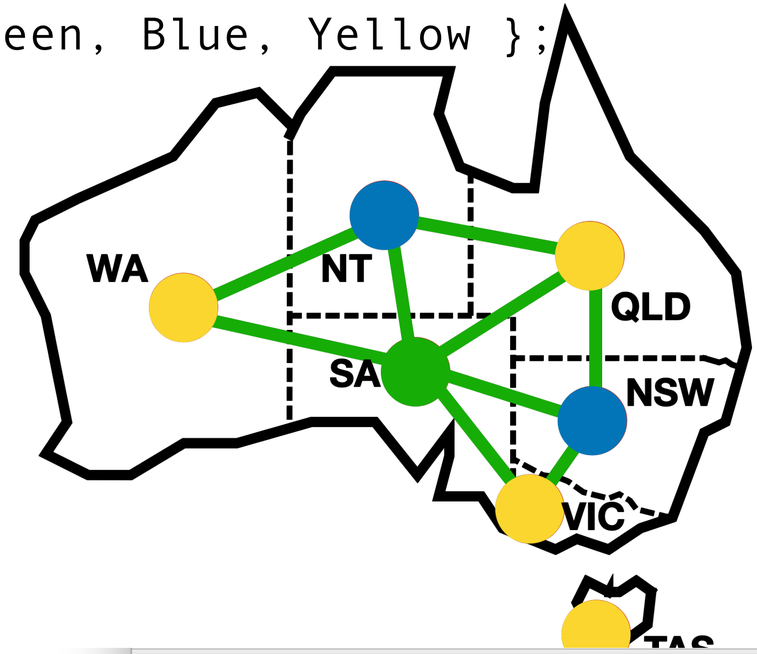
Edges connect states that share borders



# Map Colouring Model

```
enum Colour = { Green, Blue, Yellow };

var Colour: WA;
var Colour: NT;
var Colour: SA;
var Colour: VIC;
var Colour: NSW;
var Colour: QLD;
var Colour: TAS;
```




```
Running map_colouring.mzn
WA = Yellow;
NT = Blue;
SA = Green;
VIC = Yellow;
NSW = Blue;
QLD = Yellow;
TAS = Yellow;
----------
Finished in 238msec
```

Line: 1,Col: 1

# Enumerated Types

- Enumerated types ("enums") define a finite set of named objects
  - decisions and parameters may have enum types
  - very useful also for array indexes and sets
- Declared as

  `enum` *enum-name* `= {` $id_1$`, ..., ` $id_n$ `};`
- The set of names can be given separately:

  `enum` *enum-name*`;`

  *enum-name* `= {` $id_1$`, ..., ` $id_n$ `};`
- We can declare variables using enum types:

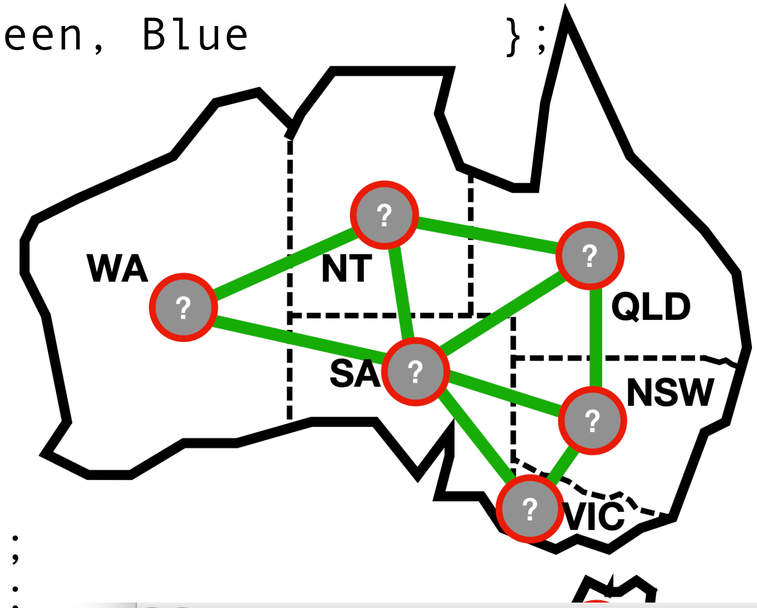
  `var` *enum-name*`:` *var-name*`;`

# What about just two colours?

```
enum Colour = { Green, Blue          };

var Colour: WA;
var Colour: NT;
var Colour: SA;
var Colour: VIC;
var Colour: NSW;
var Colour: QLD;
var Colour: TAS;

constraint WA ≠ NT;
constraint WA ≠ SA;
constraint NT ≠ SA;
constraint NT ≠ QLD;
constraint SA ≠ QLD;
constraint SA ≠ NSW;
constraint SA ≠ VIC;
constraint NSW ≠ QLD;
constraint NSW ≠ VIC;
```




```
Running map_colouring.mzn
=====UNSATISFIABLE=====
Finished in 314msec
```

Problem has no solution!

Line: 1,Col: 29

# Summary

- **Enumerated types**
  - introduce named sets of objects
  - are useful for type-safety of models
  - we'll use them a lot from now on
- **Unsatisfiable** models
  - not every problem has a solution!
  - our solvers can *prove* that there is no solution
- **Graph/map colouring**
  - a classic problem from graph theory
  - applications in compilers, timetabling, ...
  - very efficient specialised algorithms exist