Download MiniZinc

http://www.minizinc.org

Solving all the worlds (combinatorial problems

(in MiniZinc)



Peter J. Stuckey
OPTIMA and Monash University
peter.stuckey@monash.edu



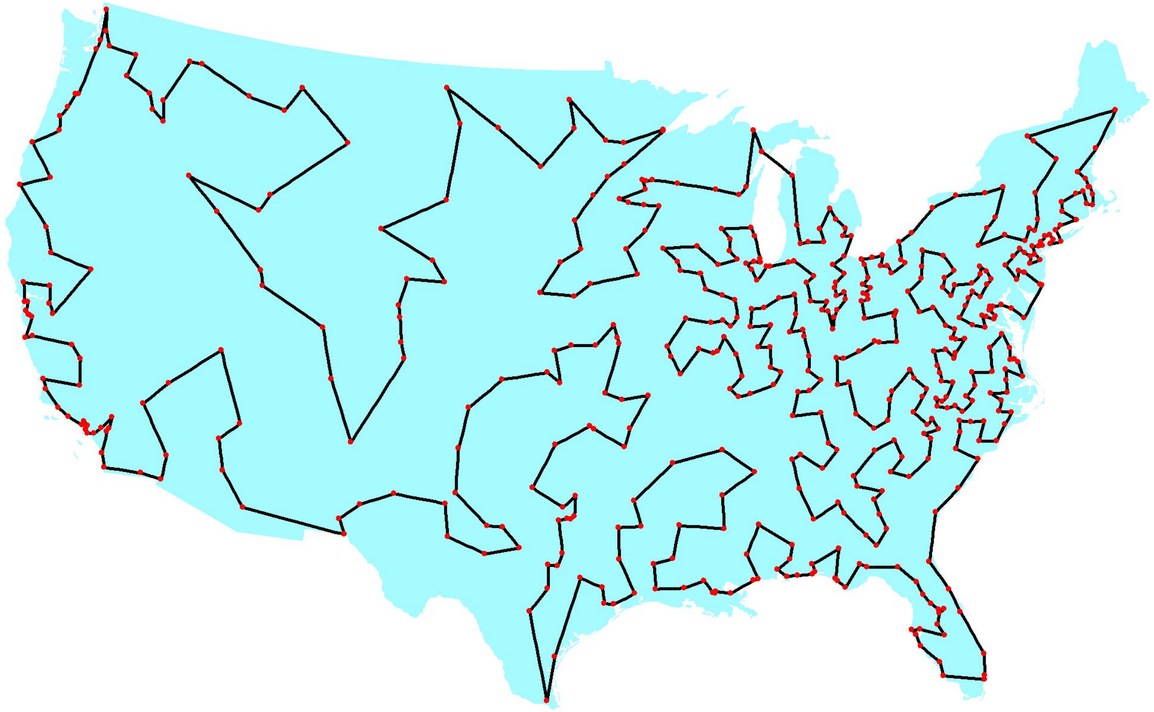


OPTiMA

ARC TRAINING CENTRE IN
OPTIMISATION TECHNOLOGIES
INTEGRATED METHODOLOGIES
AND APPLICATIONS

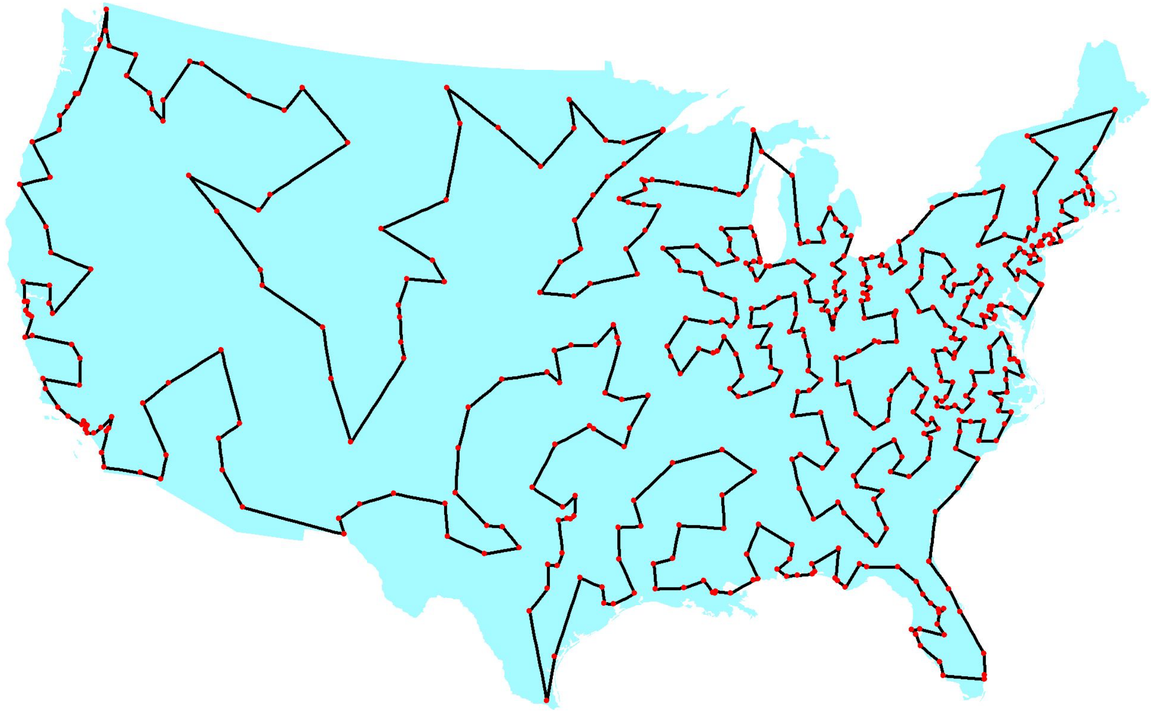
Optimisation

- Example: **Travelling salesperson**
  given a map of roads connecting cities, what is the shortest possible route visiting each city once and returning to the original city?



Quiz: *how long will it take?*

- Example: **Travelling salesperson**
  given a map of roads connecting cities, what is the shortest possible route visiting each city once and returning to the original city?



Quiz: *how long will it take?*

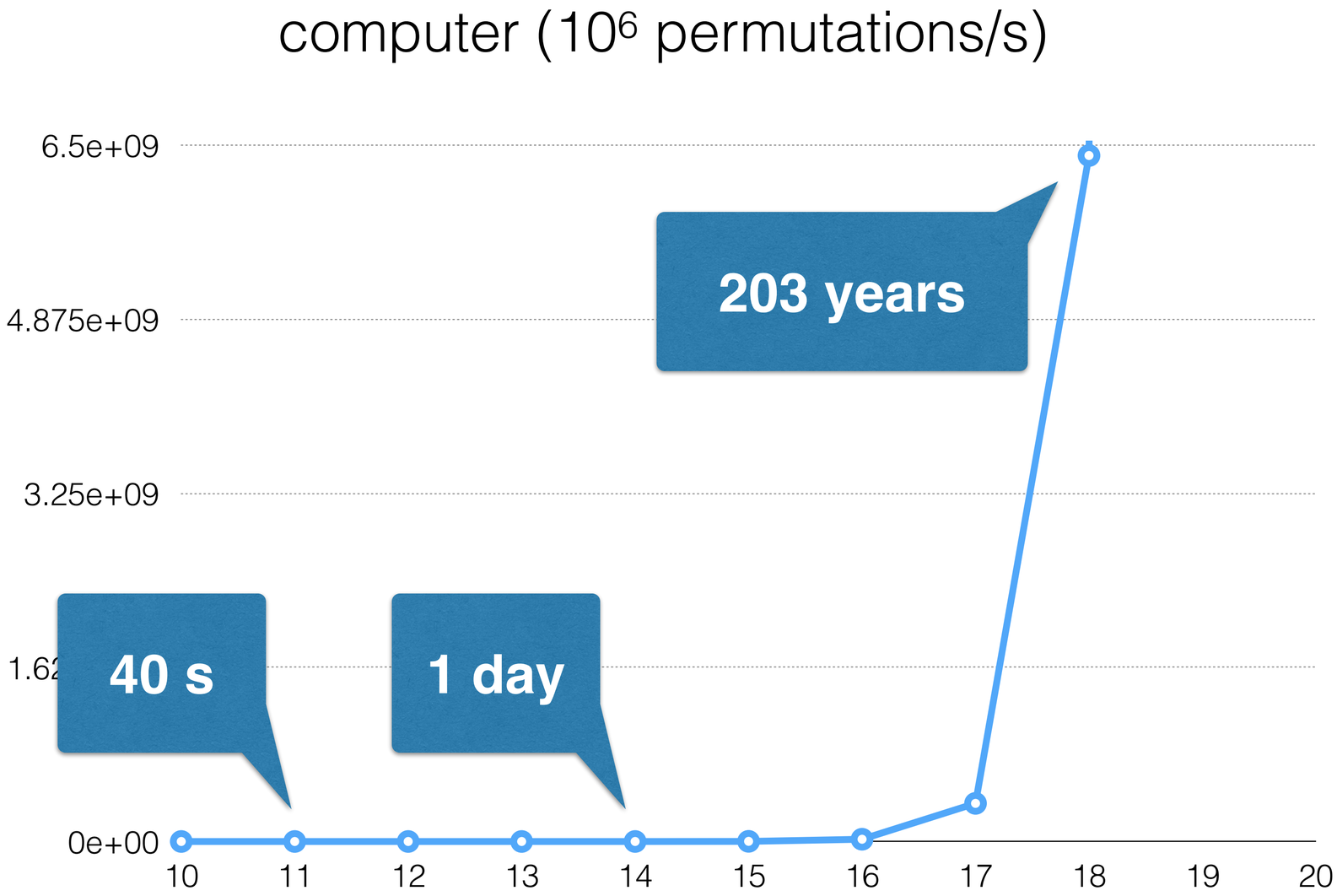
- Example: **Travelling salesperson**
  given a map of roads connecting cities, what is the shortest possible route visiting each city once and returning to the original city?

- **Brute force:** for *n* cities, check all *n!* permutations

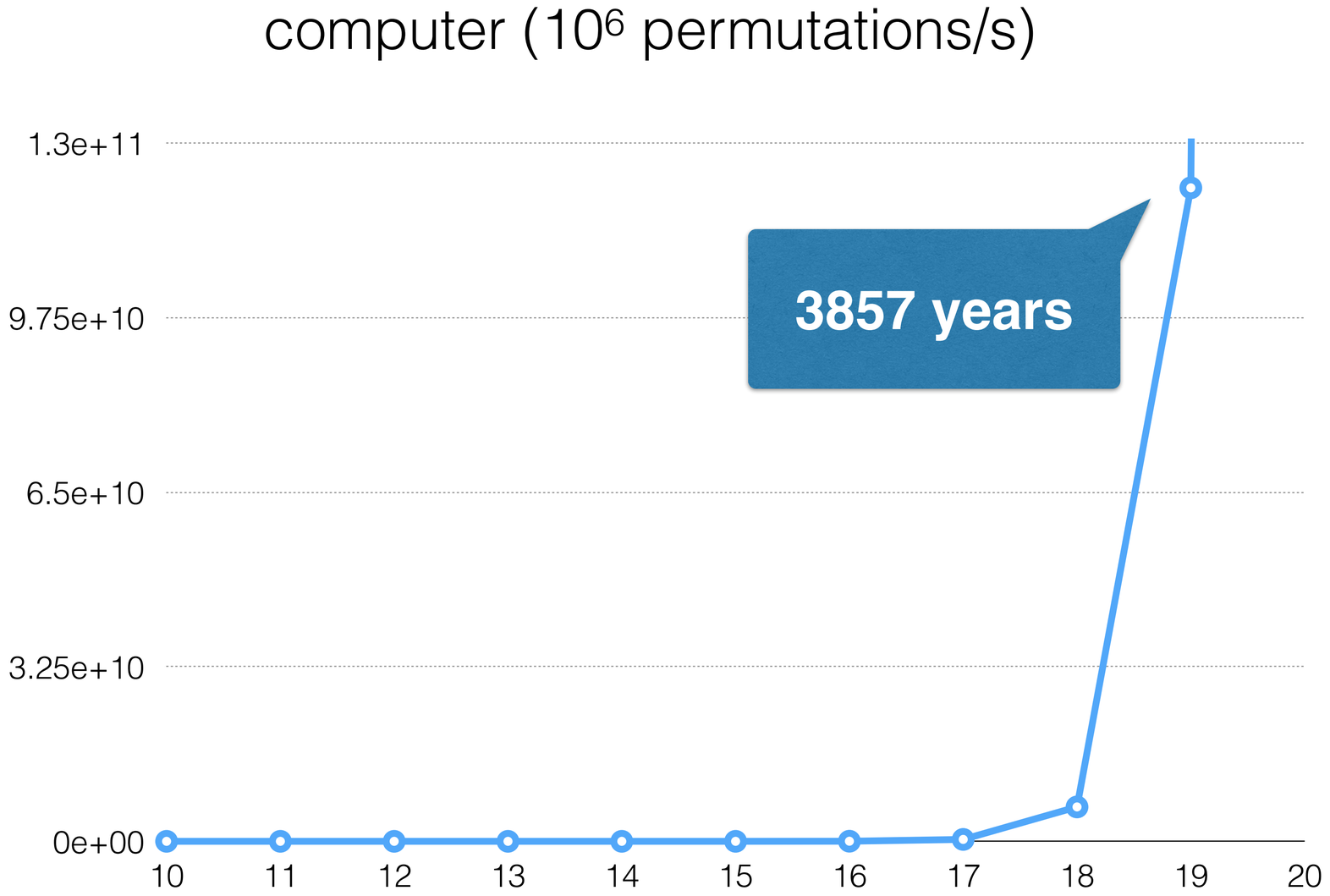
  - possible for what value of *n*? (rough guess!)

  - let's assume ~10^6 permutations per second

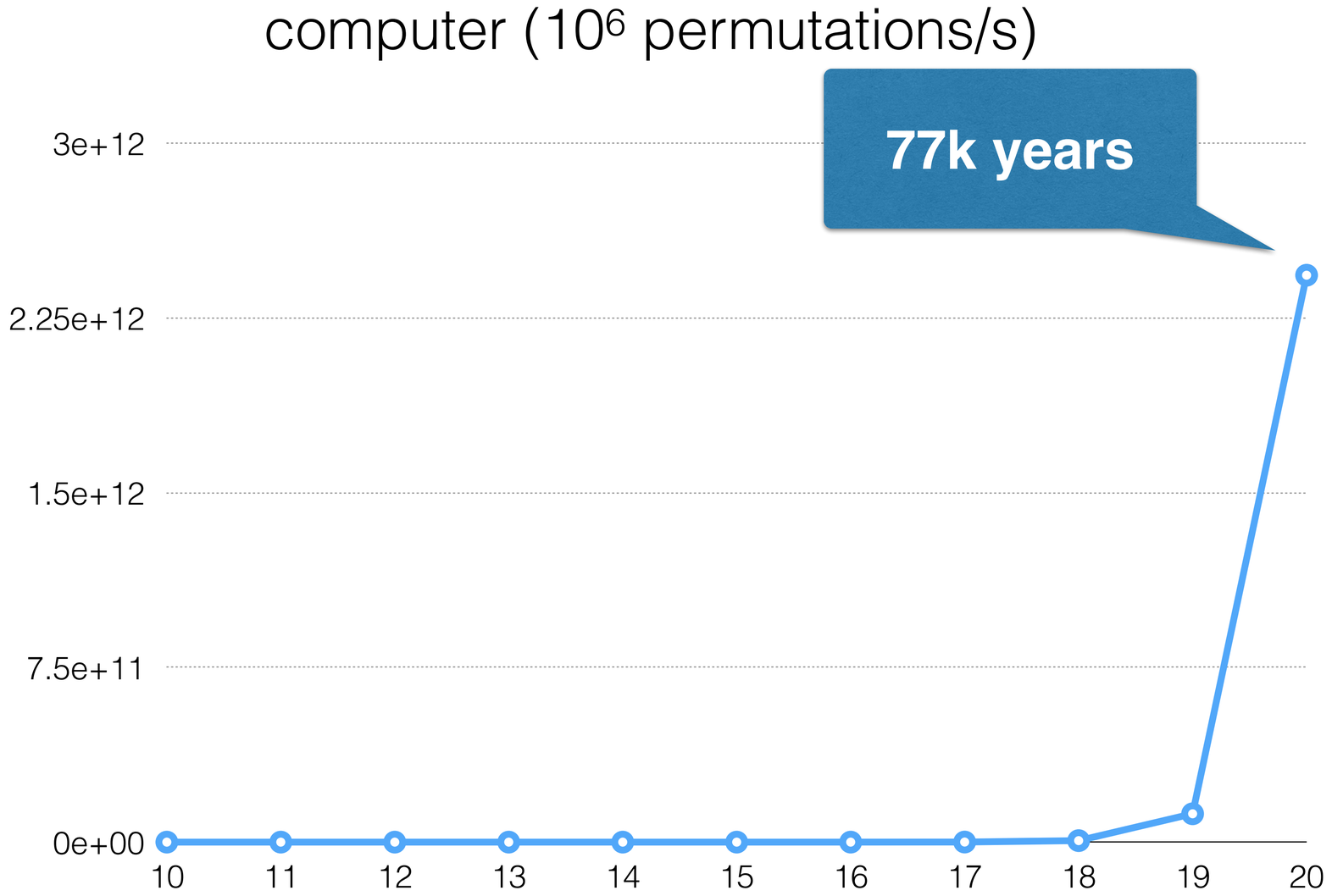
Quiz: *how long will it take?*



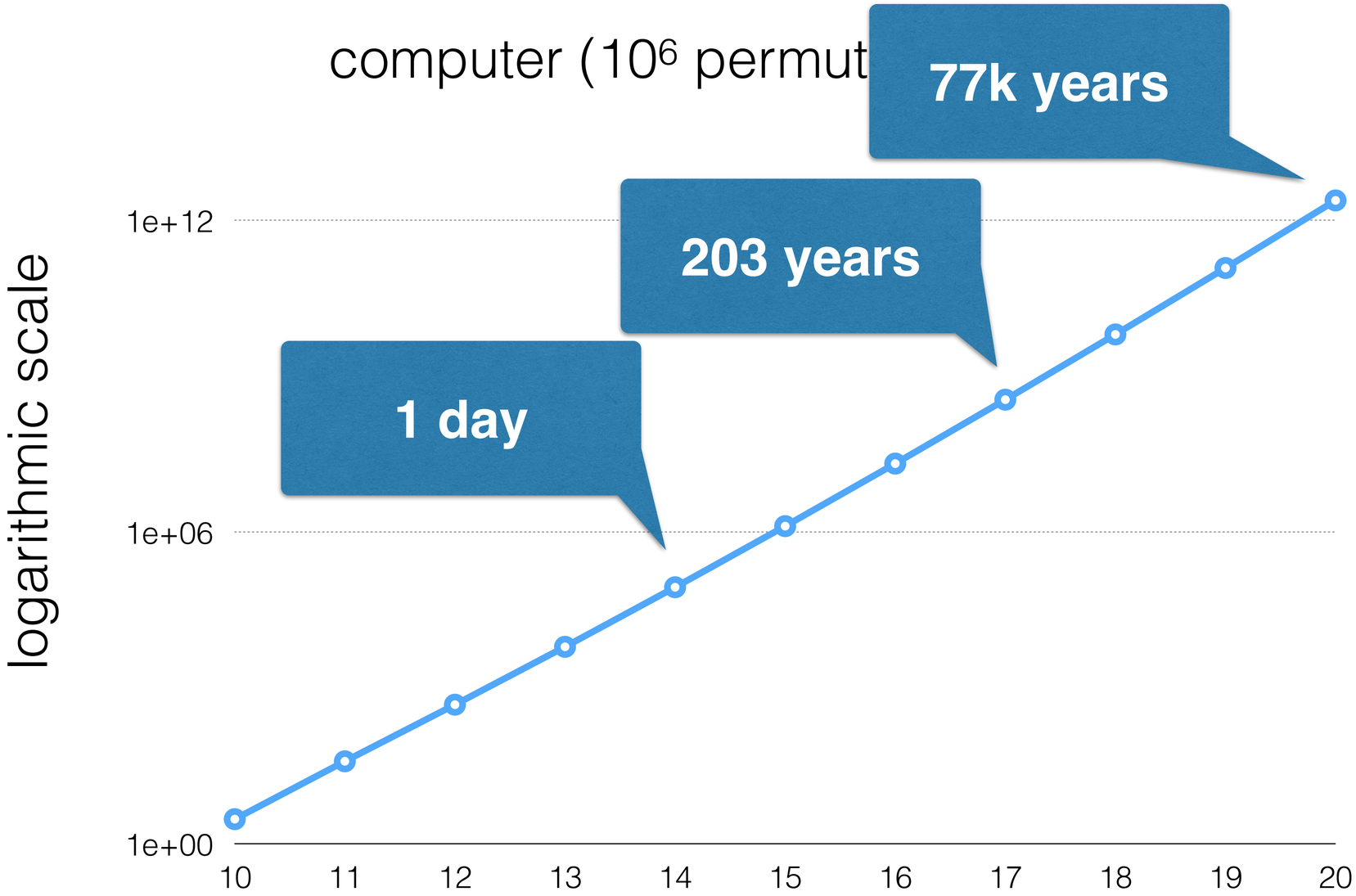
Quiz: *how long will it take?*



Quiz: *how long will it take?*



Quiz: *how long will it take?*



Quiz: *how long will it take?*

- Travelling salesperson:
  given a map of roads connecting cities, what is the shortest possible route visiting each city once and returning to the original city?

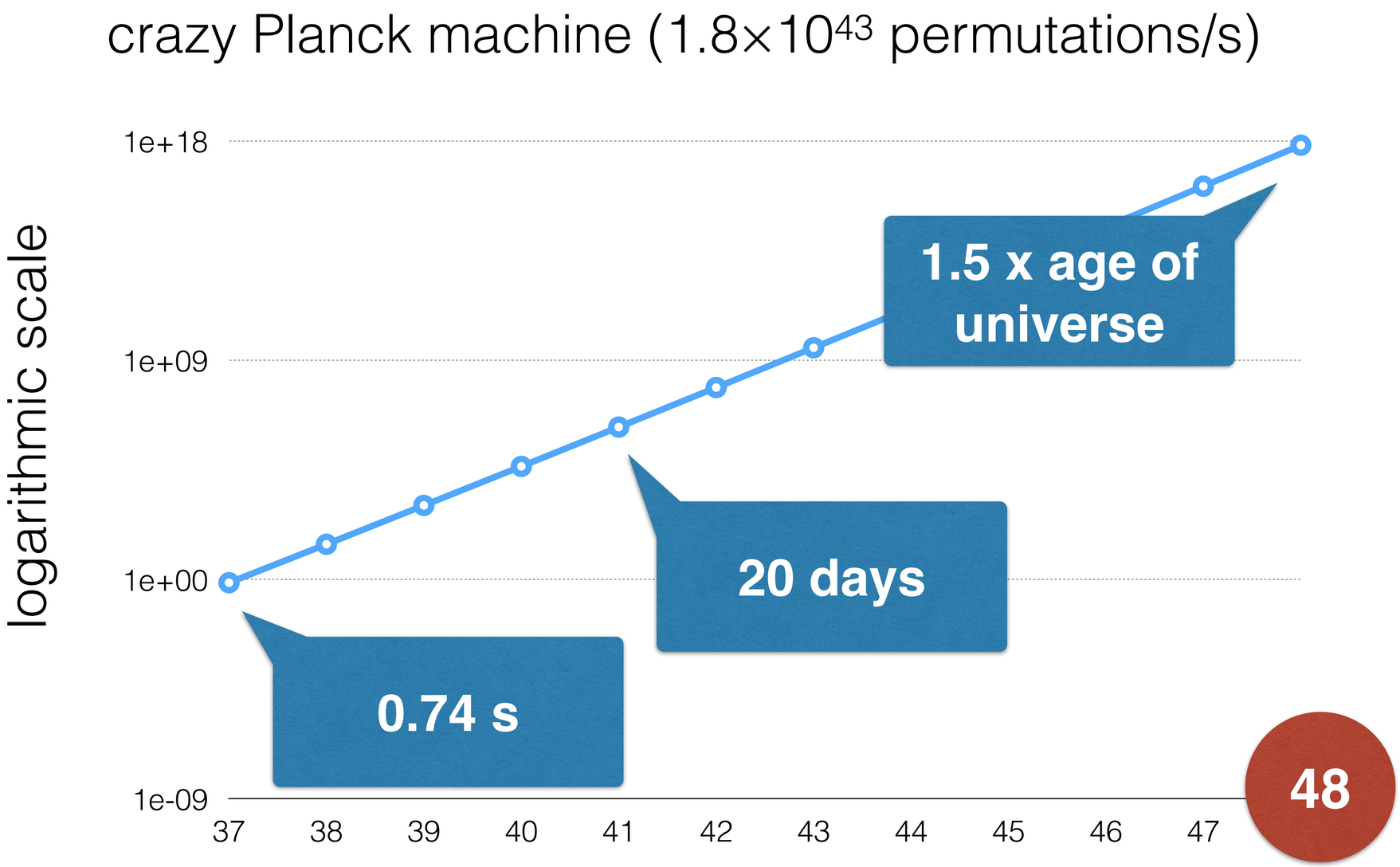
- **Brute force:** for *n* cities, check all *n!* permutations

  - possible for what value of *n*? (rough guess!)

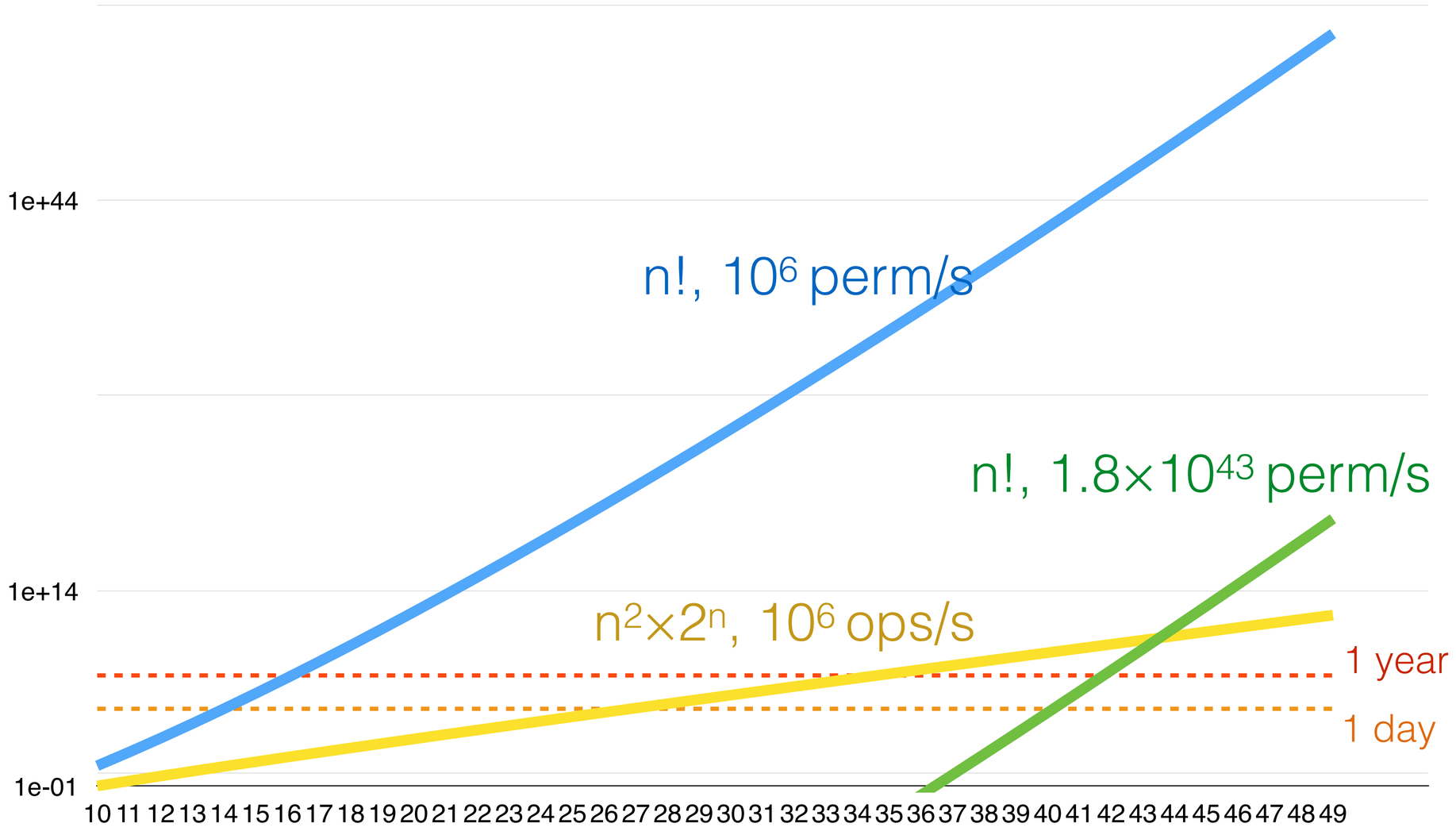
  - current hardware: maybe 10^6 permutations per second

  - how about **Planck time** (shortest "useful" time interval): $tP \approx 5.4 \times 10^{-44}$ s, so let's assume 1.8×10^{43} permutations per second

Quiz: *how long will it take?*



Quiz: *how long will it take?*



What do we learn from this?

There's always an *n* for which
all our algorithms fail.





There's an *n* up to which
our algorithms work!

The trick is to push the *n*.

(The best TSP solvers can deal with tens of thousands of cities.)

Try it yourself! http://bit.ly/2jpRr3B

- 1962 contest run by Procter&Gamble
- Find shortest tour of 33 locations in the US
- Brute force: billions of years
- **Q:** How do you do it?
  - Reasoning + educated guessing
  - Inference + search



How to push the n?

- Use advanced algorithms
- We use *generic algorithms:*
  - SAT, SMT (SAT Modulo Theories)
  - CP (Constraint Programming)
  - MIP (Mixed Integer Programming)
  - Hybrid approaches
- These can handle a **class** of problems
- Need an **encoding** of a problem **instance** as input

Model

Discrete Optimisatiom

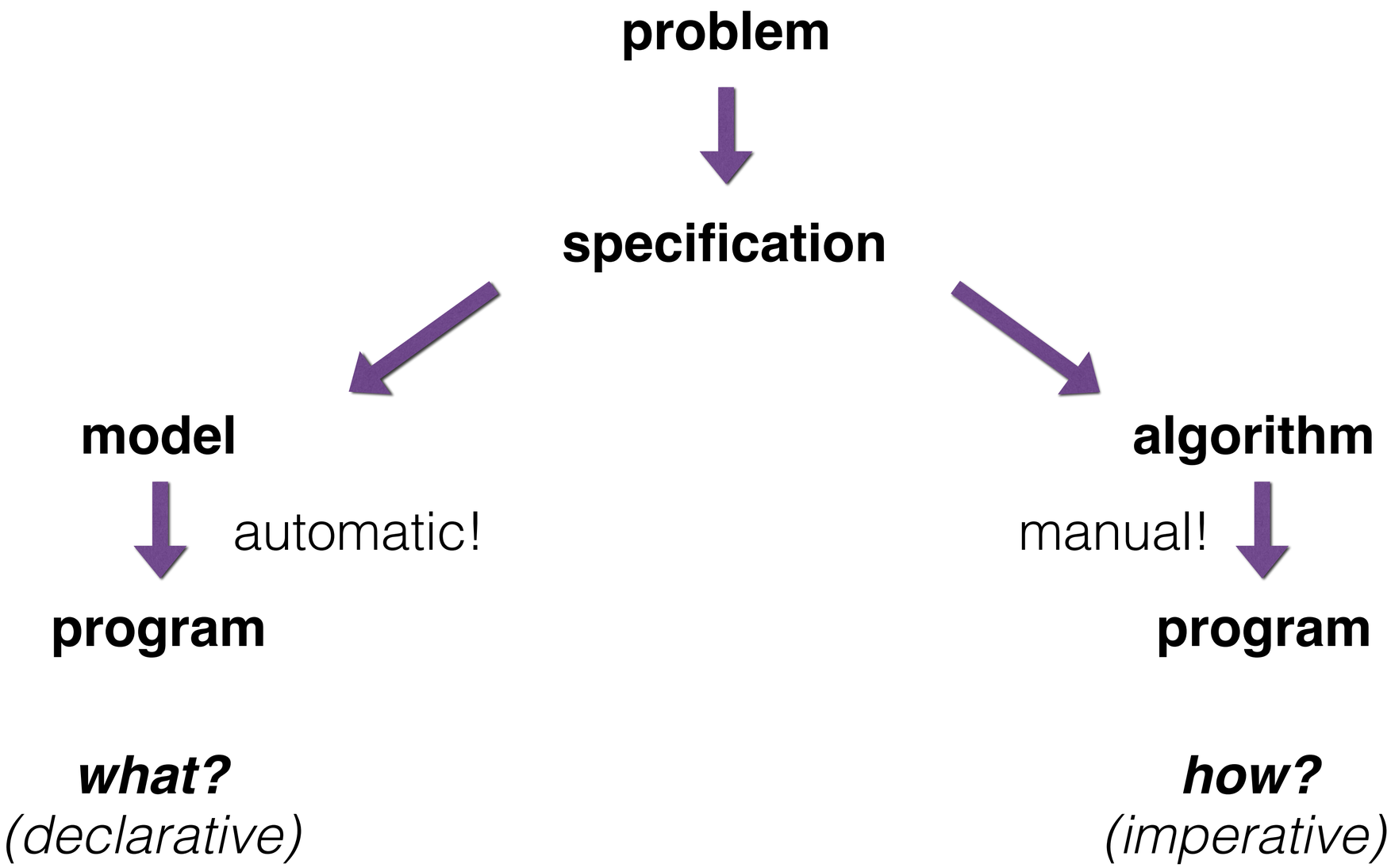
- Ubiquitous in any decision making
  - scheduling
  - rostering
  - production planning
- To solve a discrete optimisation problem, we want to
  - rapidly generate a solution to the problem of interest
  - avoid committing to the wrong approach

Why Learn Discrete Optimisation?

- Artificial Intelligence (AI) is a hot topic
  - some people seem to equate AI = Machine Learning

- But a recent McKinsey report on **State of AI in 2021** finds
  - 47% of all respondents talking about AI use cases were using **optimisation**

Modelling vs programming



Software

- We use



  www.minizinc.org
  Twitter @minizinc
  facebook.com/minizinclanguage

- Translates high-level problem specifications into efficient programs for generic solvers

- Developed by Data61, Monash University & Univ. of Melbourne

MiniZinc in the Real World

- An example from recent email

Hi Peter,

I'm just reaching out to say a big thank you for the MiniZinc courses you made available on Coursera. Using those courses as a guide I was able to design and implement a solver for a complex rail-yard scheduling problem for Aurizon Network that is now used daily for planning.

Map Colouring

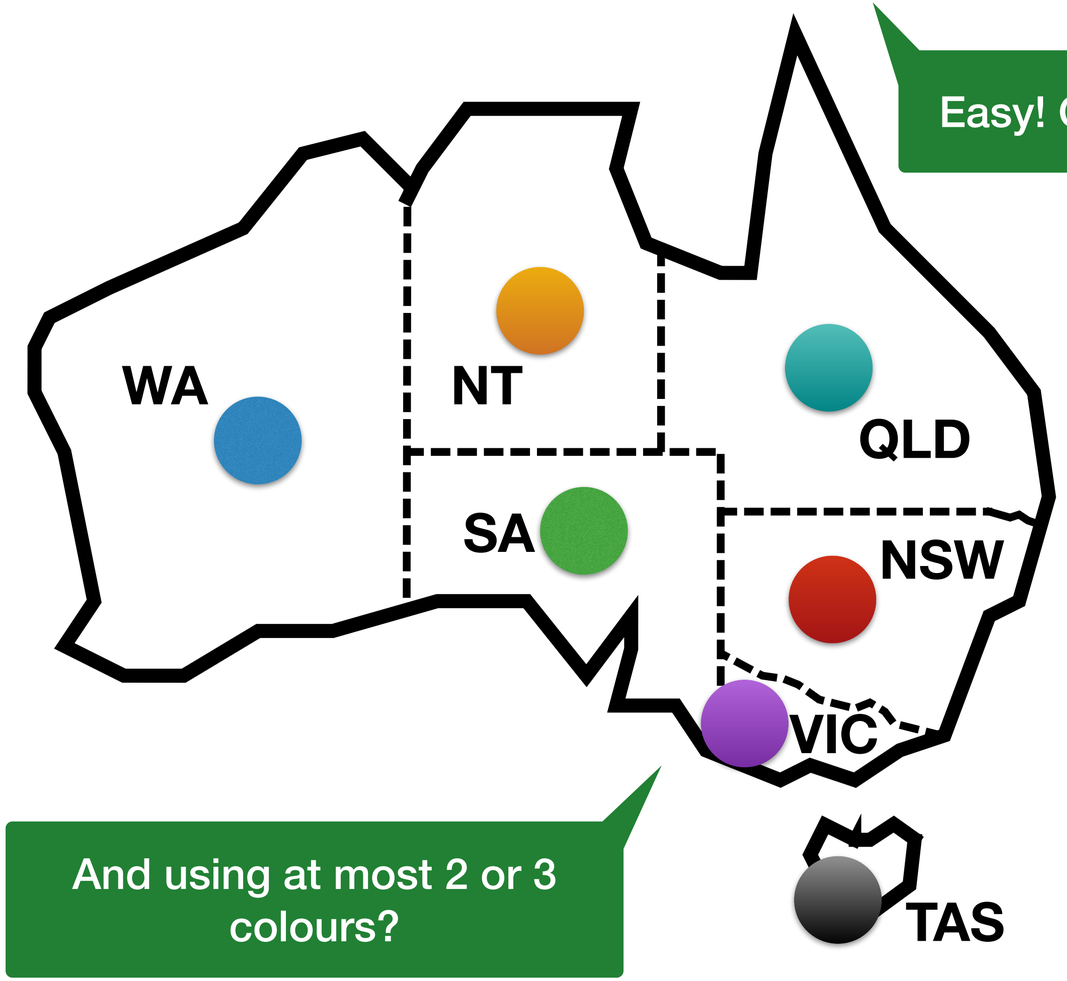
Peter Stuckey



MONASH
University

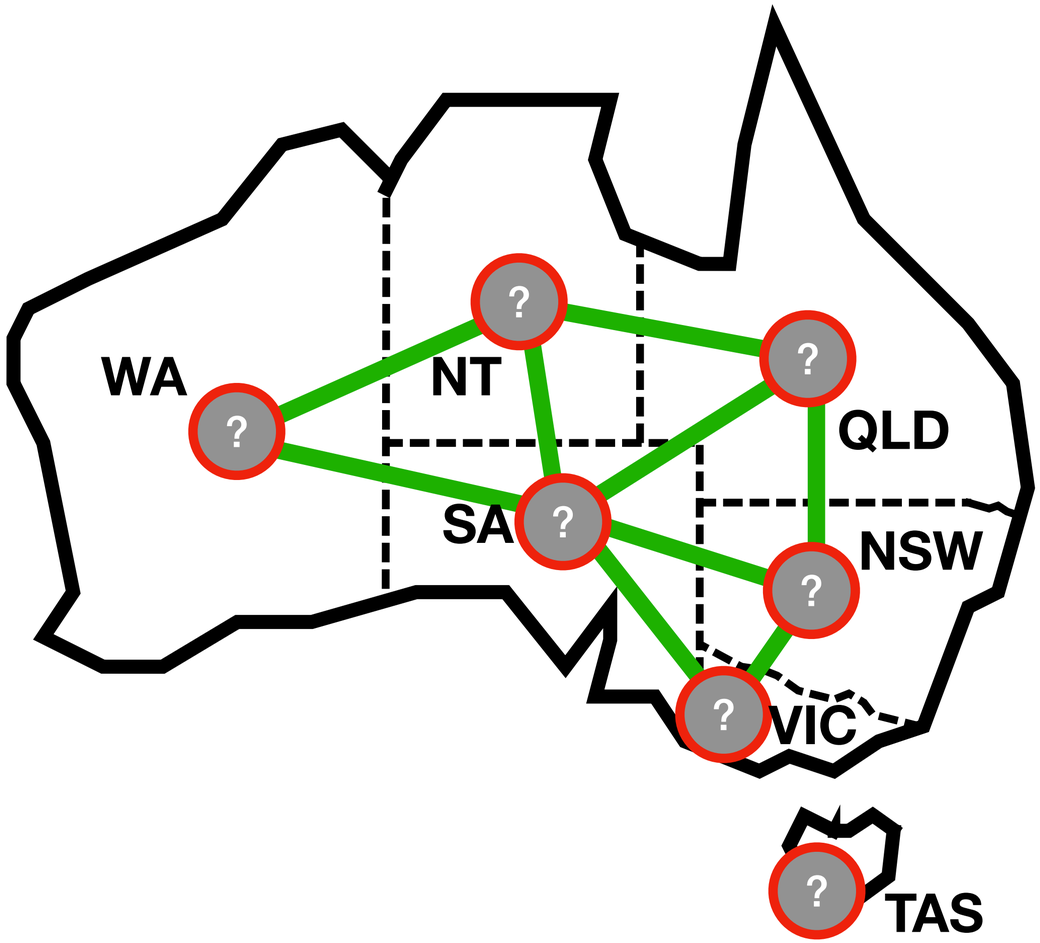
Map Colouring

Goal: colour adjacent states with different colours



Map Colouring is Graph Colouring

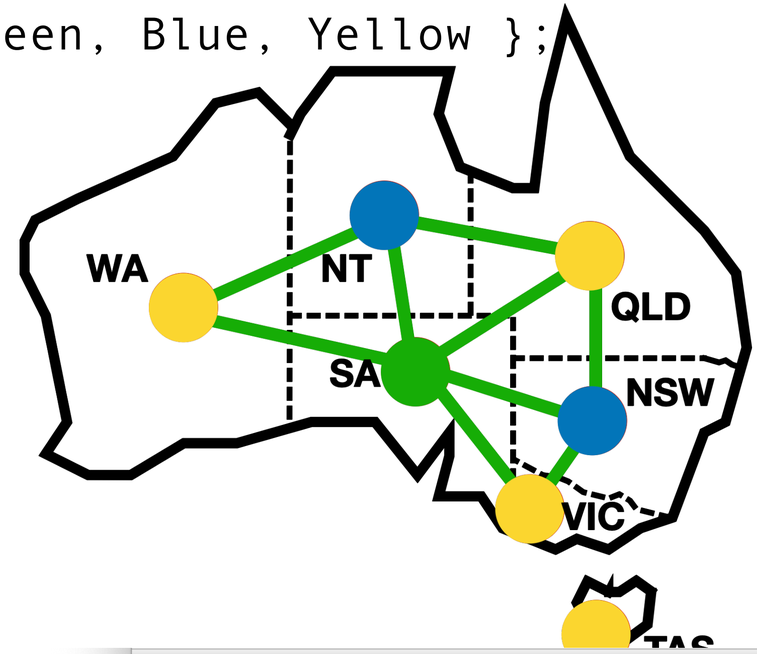
Edges connect states that share borders



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:

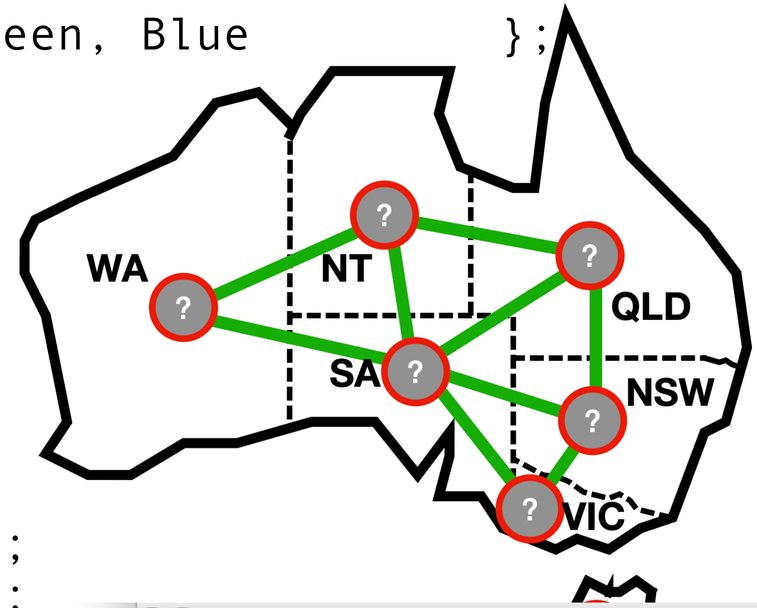
  `var` *enum-name*`:` *var-name*`;`

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

Buying Robots

Peter Stuckey

MONASH
University

Setting Up the Toy Shop



	A	B	C	D
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?**



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

Mathematical model

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

linear objective

$$\textit{subject to}$$

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

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

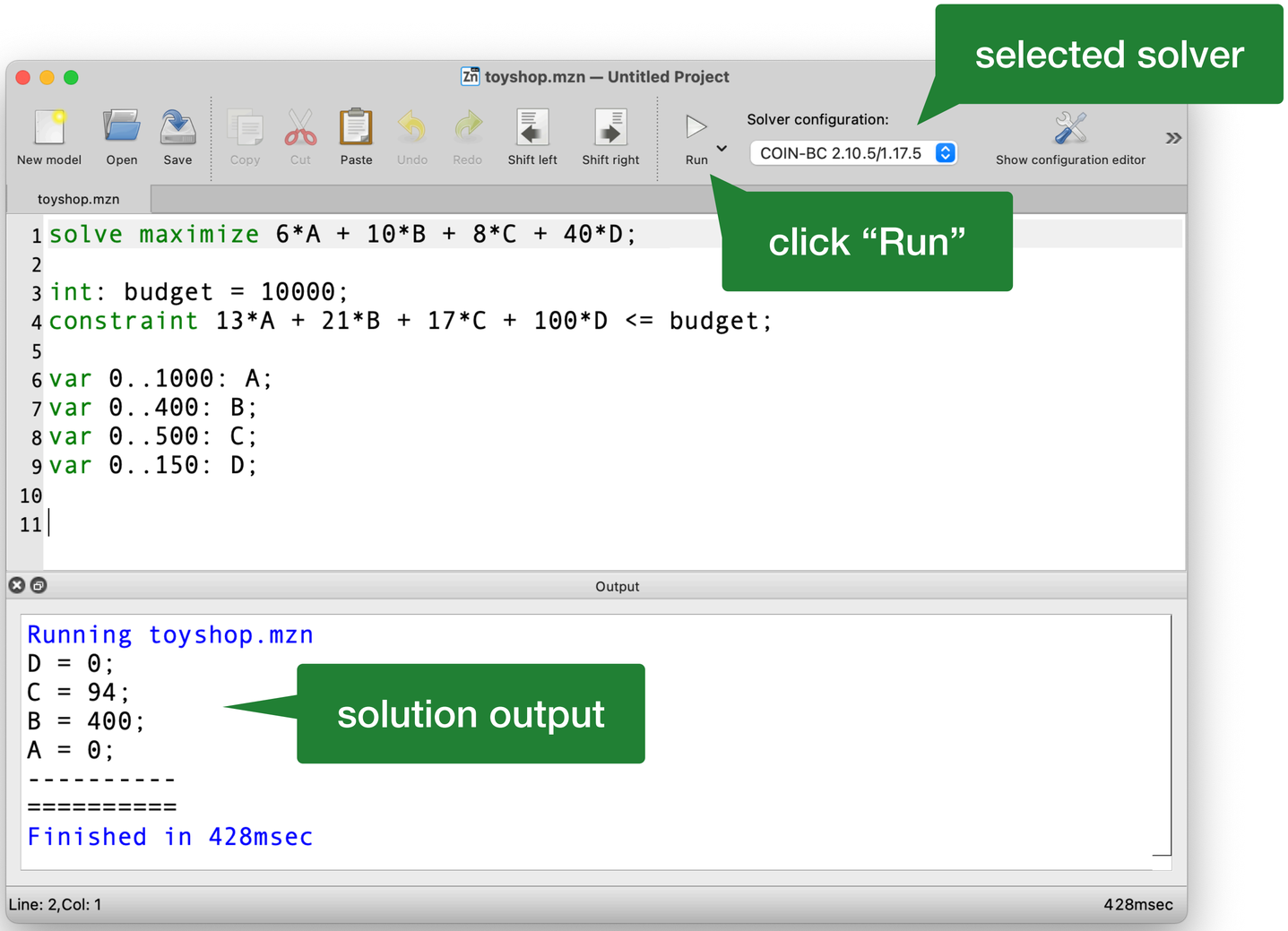
  `constraint` *`<constraint-expression>`*

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

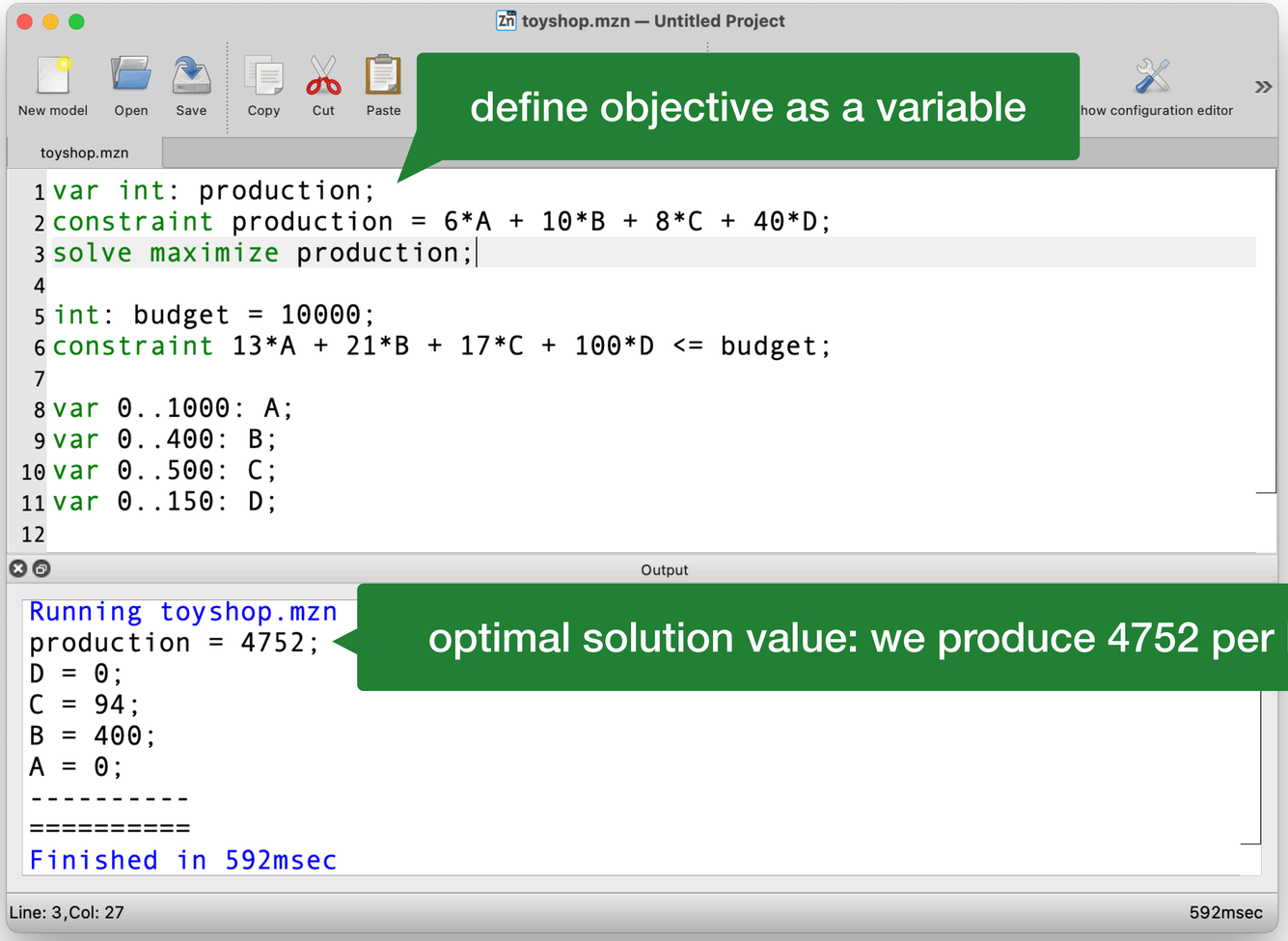
```
Running toyshop.mzn
D = 0;
C = 94;
B = 400;
A = 0;
----------
==========
Finished in 428msec
```

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!

MiniZinc model structure

- Sequence of **items**
  - Inclusion: `include "file.mzn"`
  - Variable declaration: `var <type>: x;`
  - Variable assignment: `x = <expression>;`
  - Constraint: `constraint <Boolean expression>;`
  - Solve item (at most one per model):
    `solve satisfy;` (default meaning if none)
    `solve minimize <arithmetic expression>;`
    `solve maximize <arithmetic expression>;`
  - Predicate and test items
  - Annotation items
- Order of items does not matter

Data and Instances

Peter Stuckey

MONASH
University

Toy Shop MiniZinc model (revisited)

```
int: budget;

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

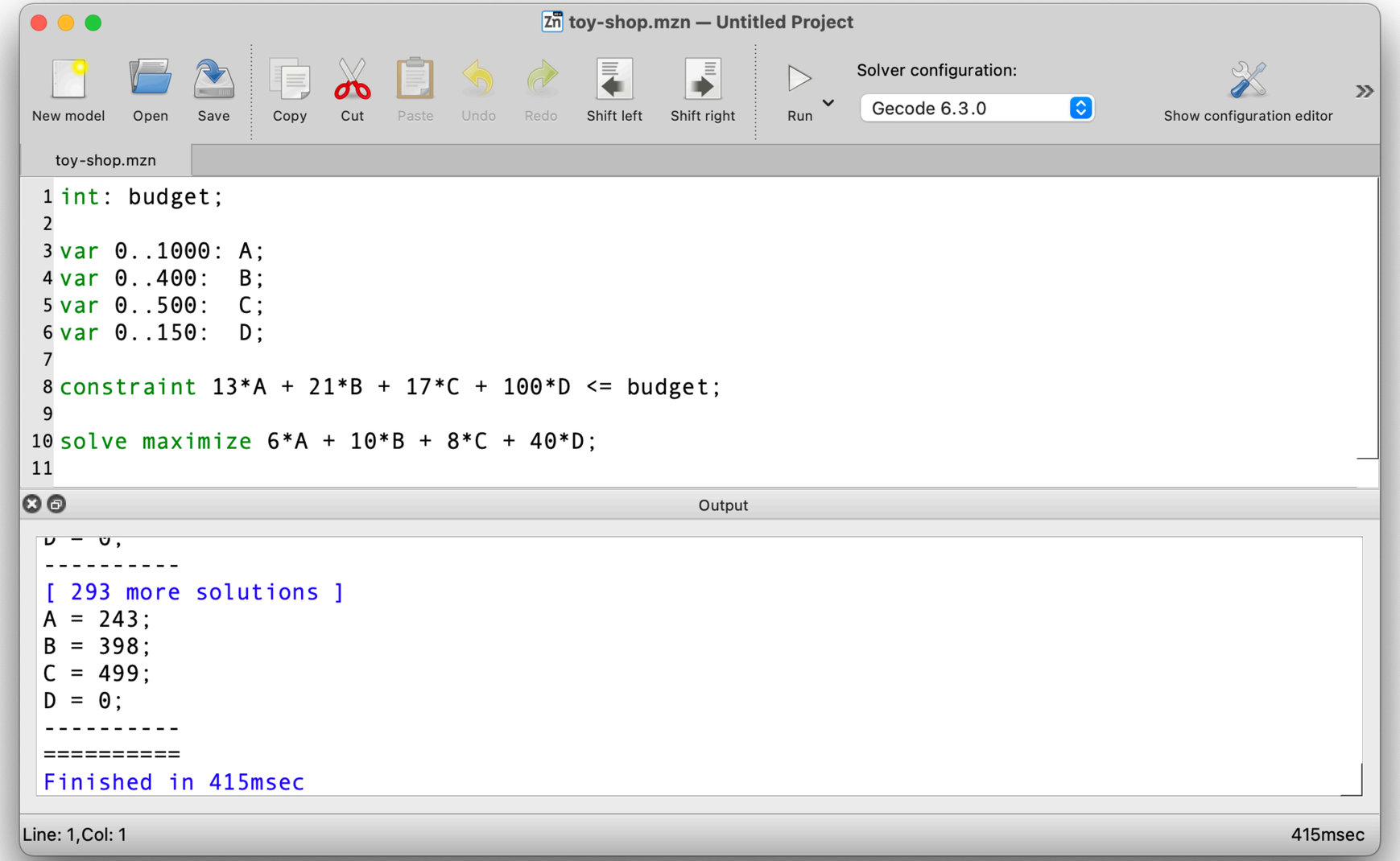

constraint
      13*A + 21*B + 17*C + 100*D <= budget;



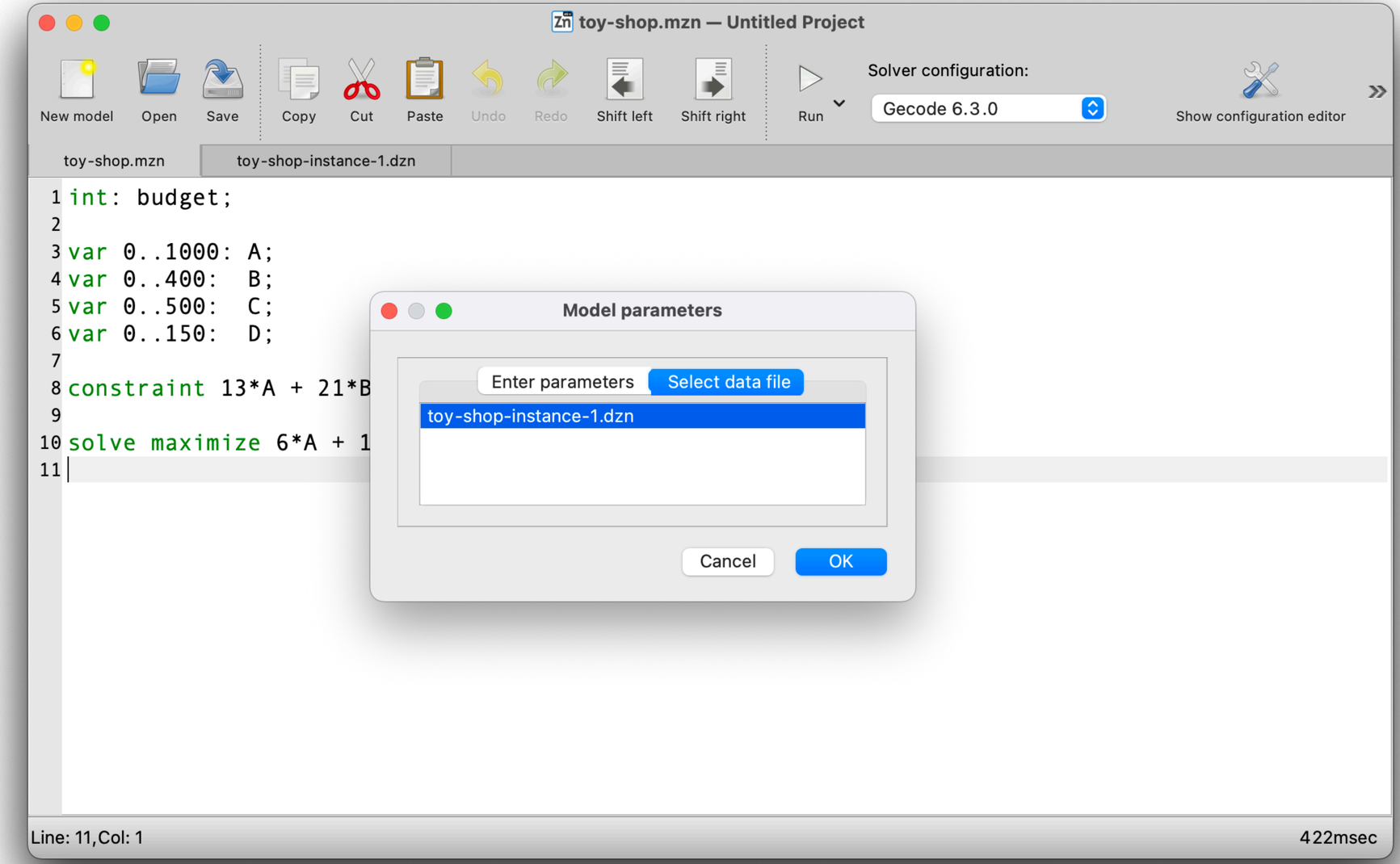
solve maximize 6*A + 10*B + 8*C + 40*D;
```

no value given!

Parameter values - IDE



Data files


toy-shop.mzn — Untitled Project
New model
Open
Save
Copy
Cut
Paste
Undo
Redo
Shift left
Shift right
Run
Solver configuration:
Gecode 6.3.0
Show configuration editor
toy-shop.mzn
toy-shop-instance-1.dzn
1 int: budget;
2
3 var 0..1000: A;
4 var 0..400: B;
5 var 0..500: C;
6 var 0..150: D;
7
8 constraint 13*A + 21*B
9
10 solve maximize 6*A + 1
11
Model parameters
Enter parameters
Select data file
toy-shop-instance-1.dzn
Cancel
OK
Line: 11,Col: 1
422msec

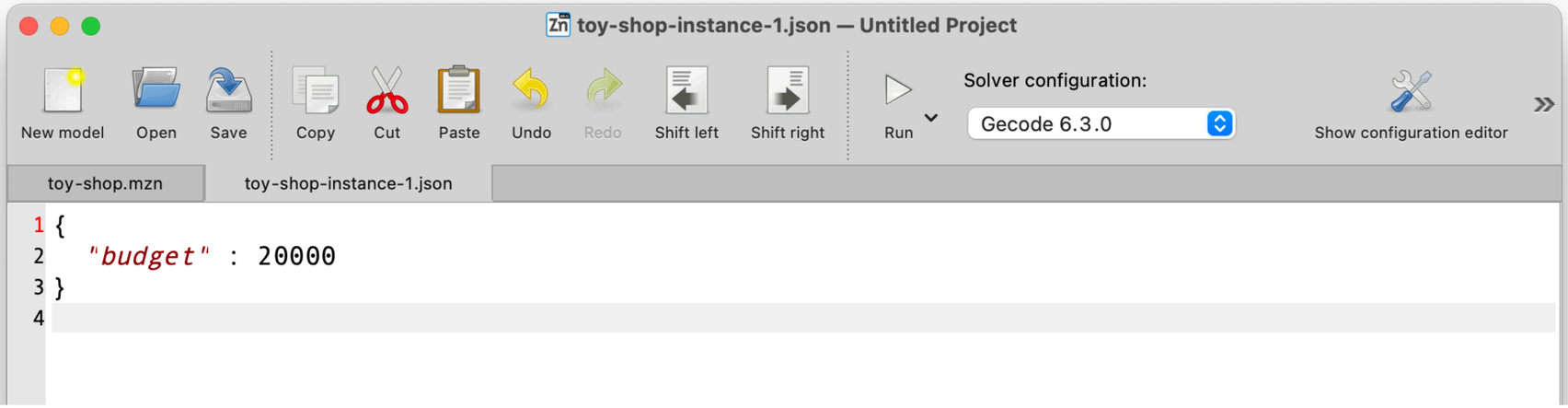
What's in a data file?

- Data files contain the values for model parameters
- All model parameters must be assigned before solving
  - in the model, in one or more data files, or as immediate parameters (IDE or `-D` command line flag)
- Can also be the values of an enumerated type, e.g. for `color.mzn`:

```
enum Color;                 (in color.mzn)

Color = {R,W,B,G,P};        (in color.dzn)
```

JSON data *(optional)*



- MiniZinc supports JSON for data input
- Can make it easier to interface to other systems
- Format: object where each field is a parameter
- Supports simple values, lists, sets, etc (see documentation)

Command line use *(optional)*

- You can give one or more data files as an additional arguments:

```
$ minizinc toy-shop.mzn toy-shop-data.dzn
```

- Or specify assignments directly

```
$ minizinc toy-shop.mzn -D"budget=10000;"
```

Summary

- A **model** is a formal description of an entire class of problems
- An **instance** is one particular problem
- To turn a model into an instance:
  - provide values for all parameters
- In MiniZinc, parameter values are usually provided in **data files** (`.dzn` or `.json`)

Arrays

Peter Stuckey



MONASH
University

The Toy Shop Planner

- We only solved a single instance of this problem
- Everything was fixed:
  - the productivity, the cost per robot, the number of robots available
  - and, crucially: **the set of robot types**!
- What if we want to **reuse** the same model for different instances?



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

The Toy Shop Planner

```
enum ROBOT;
int: budget;

array[ROBOT] of int: available;
array[ROBOT] of int: cost;
array[ROBOT] of int: tph; % toys per hour

% how many robots to buy
arr                          .inf

constraint
    forall(r in ROBOT) (buy[r] <= available[r]);

constraint
    sum(r in ROBOT) (cost[r]*buy[r]) <= budget;

solve maximize
    sum(r in ROBOT) (tph[r]*buy[r]);
```

array declarations

forall expression

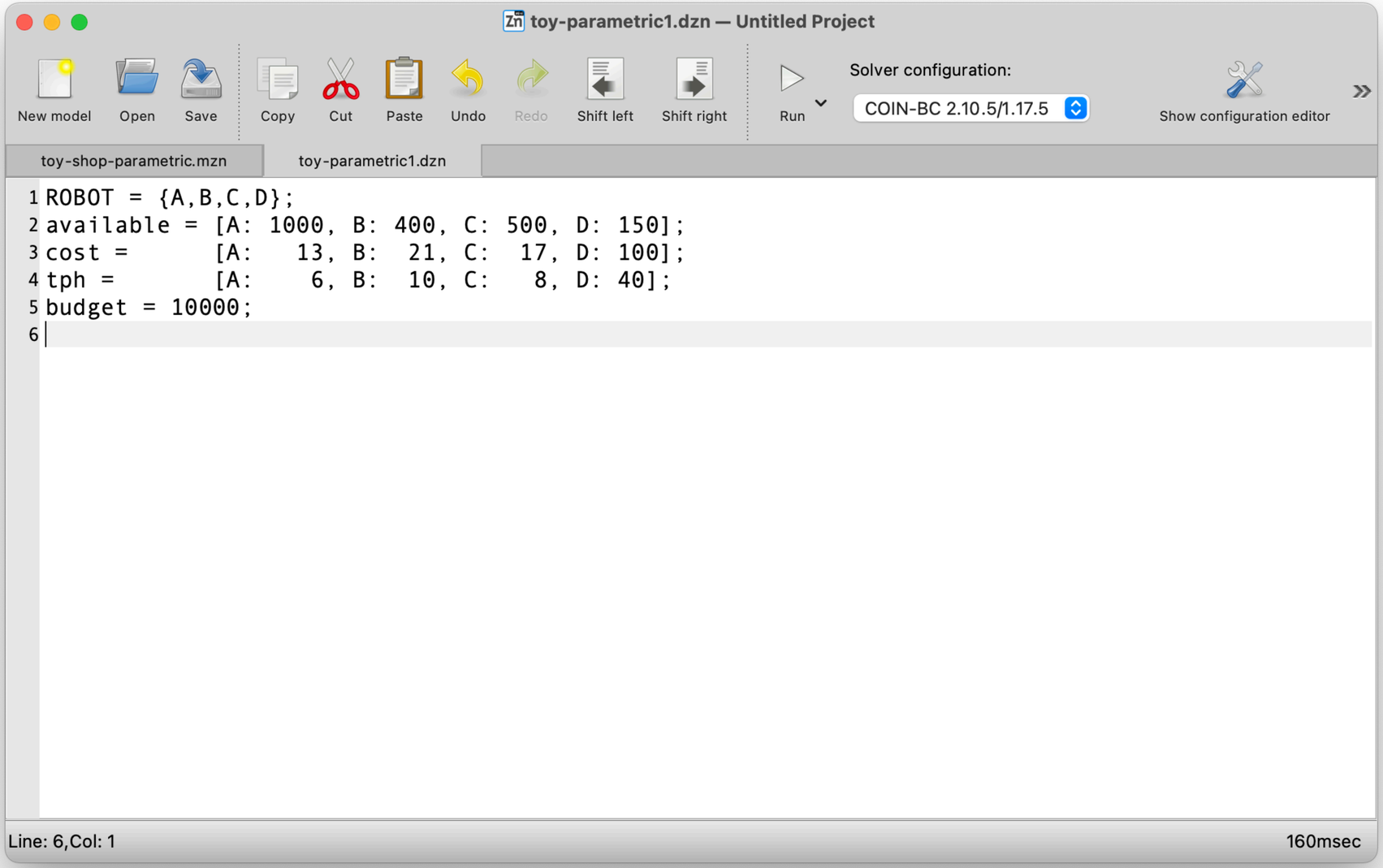
array lookup

sum over an array

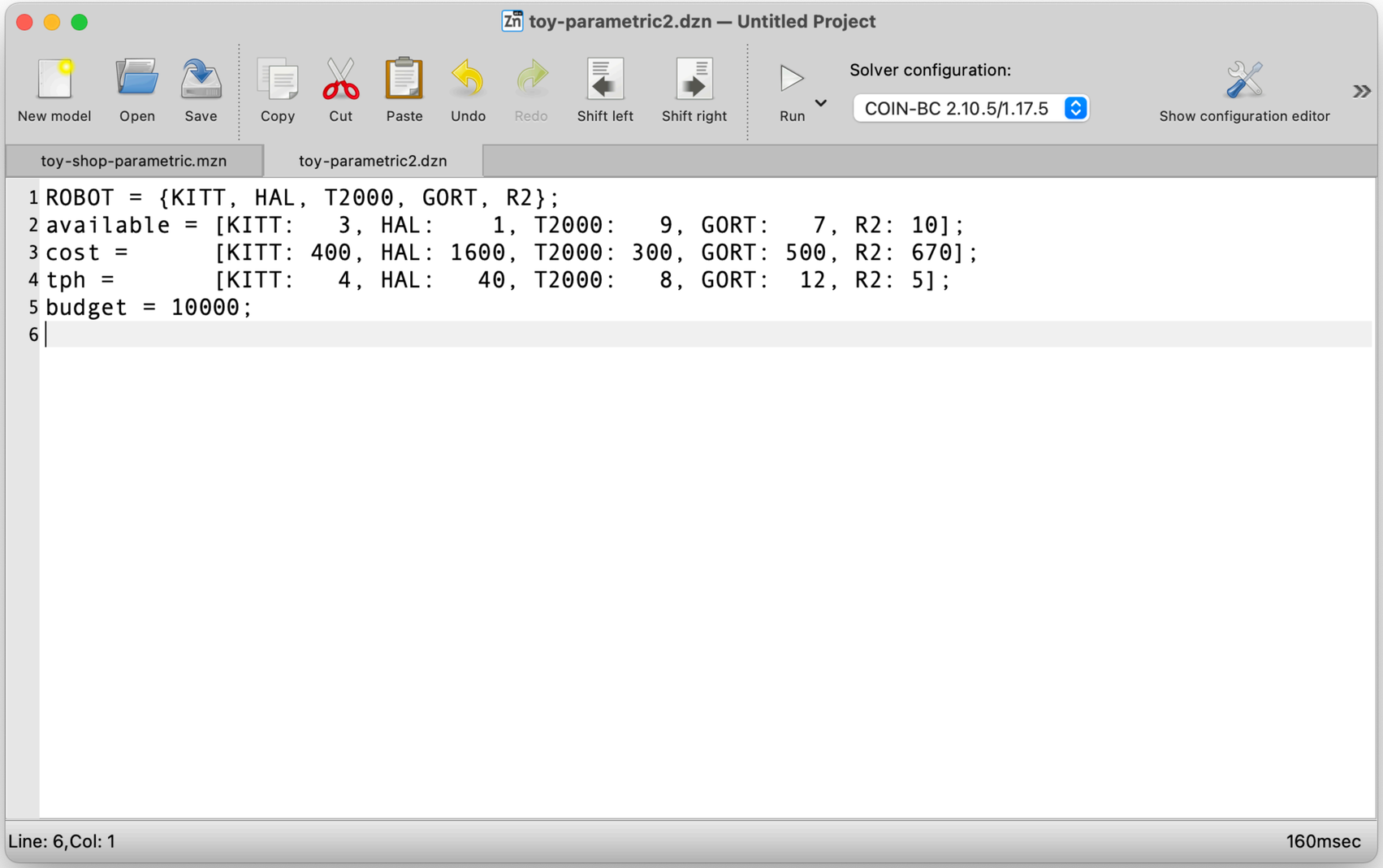
Arrays and Generators

- Arrays can hold parameters and variables
  `array[`*`range`*`] of `*`variable declaration`*
- Range expressions:
  `l..u` (where `l`, `u` are integers), or enumerated type
- Array lookup
  `array-name``[`*`index-expression`*`]`
- Generator expressions
  - `forall(i in range) (`*`bool-expression`*`)`
    for all `i` in the range, *`bool-expression`* is true
  - `sum(i in range) (`*`expression`*`)`
    sum over all *`expression`* for all `i` in the range

Data files with arrays (toy-parametric1.dzn)



Data files with arrays (toy-parametric2.dzn)



Modelling Objects

- Create an **enumerated type** with the object names (e.g. `ROBOT`)
- Create **parameter arrays** for each attribute of the object (e.g. `available`, `cost`, `tph`)
- Create **variable arrays** for each decision of the objects (e.g. `buy`)
- Use **generators** to add constraints over the objects (e.g. `forall`, `sum`)
- Models often have multiple sets of objects

Summary

- Use enumerated types to represent **sets of objects**
- Arrays represent objects **attributes** and **decisions**
- **Generator expressions** are used to construct expressions over multiple objects
- The toy shop planner problem is a version of the well-known **knapsack problem**

Production Planning

Peter Stuckey



MONASH
University

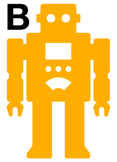
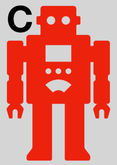
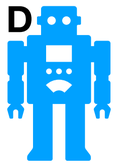
Production Planning

- Now that we've bought the robots, let's plan the actual toy production.
- We can produce 5 types of toys, each with an associated profit:

toy:					
profit:	11	18	15	17	11

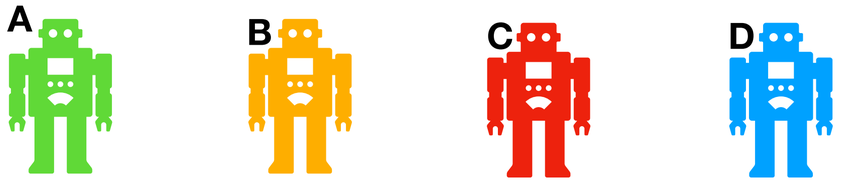
Production Planning

- Producing each toy needs a certain amount of time of each robot:

					
A 	1.5	2.0	1.5	0.5	0.1
B 	1.0	0.0	0.5	1.0	2.5
C 	1.0	2.0	1.0	0.9	0.1
D 	1.0	0.0	1.0	1.5	2.5

Production Planning

- And each robot has a limited production capacity:



5000 7500 4000 3000

Production Planning

- This problem is very similar to earlier problems
  - We have "products" to make
  - Each product consumes some resources
  - There are limits on resources
  - We want to maximise profit
- Original toy shop problem: resource = budget, product = robots, profit = toys per hour
- A generic model can capture all kinds of production planning problems

Production Planning: Parameters

```
% products
enum PRODUCT;
% Profit per unit for each product
array[PRODUCT] of float: profit;

% resources
enum RESOURCE;
% Amount of each resource available
array[RESOURCE] of float: capacity;

% Units of each resource required to produce
%       1 unit of product
array[RESOURCE,PRODUCT] of float: consumption;
```

two-dimensional array

Production Planning: Constraints

```
% Variables: how much to make of each product
array[PRODUCT] of var 0..infinity: produce;


% Production can only use the available resources:
constraint forall(r in RESOURCE)(
  sum (p in PRODUCT)
      (consumption[r,p] * produce[p]) <= capacity[r]
);


% Maximize profit
var int: objective;
constraint objective= sum(p in PRODUCT)
   (profit[p]*produce[p]);
solve maximize objective;
```

two-dimensional array lookup

Arrays

- Arrays can have multiple dimensions:

  `array[`*`index_set_1`*`, `*`index_set_2`*`, ...] of `*`type`*

- Each index set needs to be
  - an integer range (e.g. `1..10`)
  - an enumerated type (e.g. `PRODUCT`)
  - or a fixed set expression whose value is a range (e.g. `min(S)..max(T)`)
- Arrays can contain any type **except** other arrays

Array literals

- 1-dimensional arrays are initialised using a list, e.g.

```
profit = [400, 500];
capacity = [4000, 6, 2000, 60, 50];
```

- We have special syntax for 2-dimensional arrays:

```
consumption = [| 1.5, 1.0, 1.0, 1.0
               | 2.0, 0.0, 2.0, 0.0
               | 1.5, 0.5, 1.0, 1.0
               | 0.5, 1.0, 0.9, 1.5
               | 0.1, 2.5, 0.1, 2.5 |];
```

start with `[|`

separate rows with `|`

end with `|]`

Indexed Array Literals

- We can also specify the indices directly
- For 1d literals:

```
capacity = [R1: 4000, R2: 6, R3: 2000,
            R4: 60, R5: 50];
```

- For 2d literals:

```
consumption =
  [|              A:   B:   C:   D:
   | Duck:        1.5, 1.0, 1.0, 1.0
   | Rattle:      2.0, 0.0, 2.0, 0.0
   | PiggyBank:   1.5, 0.5, 1.0, 1.0
   | Puzzle:      0.5, 1.0, 0.9, 1.5
   | Balloon:     0.1, 2.5, 0.1, 2.5 |];
```

Array index set coercion

- We can *coerce* any array into an array with a different index set, e.g. 1D to 2D:

```
consumption = array2d(1..5, 1..4,
    [1.5, 1.0, 1.0, 1.0,
     2.0, 0.0, 2.0, 0.0,
     1.5, 0.5, 1.0, 1.0,
     0.5, 1.0, 0.9, 1.5,
     0.1, 2.5, 0.1, 2.5];
```

index sets of resulting array

- Similar functions exist for 1 to 6 dimensions:

```
array1d, array2d, array3d, array4d,
array5d, array6d
```

- Error if the coerced array has the wrong size

Production Planning: Data

```
PRODUCT = { Duck, Rattle, PiggyBank,
            Puzzle, Balloon };

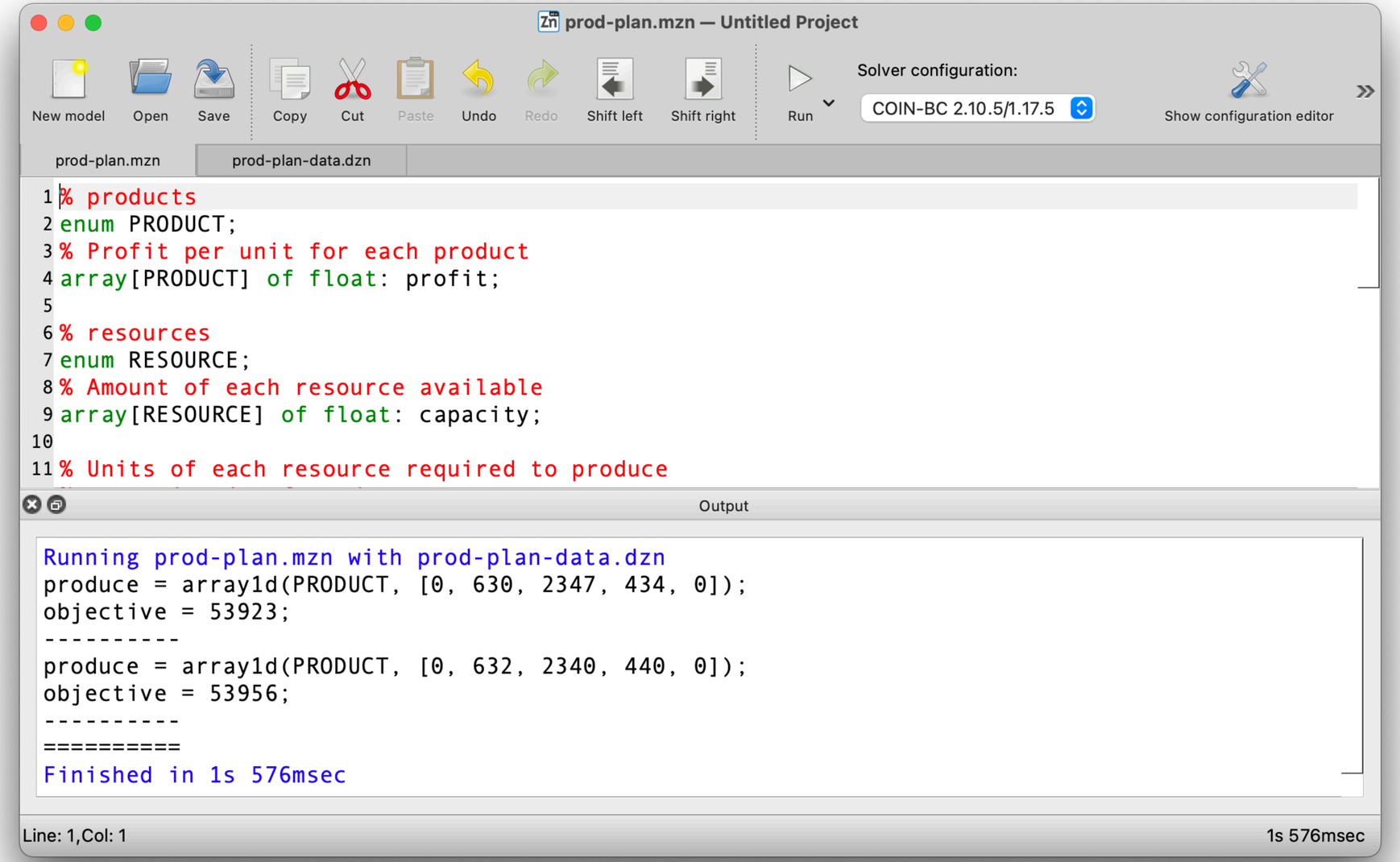
RESOURCE = { A, B, C, D };

capacity = [5000, 7500, 4000, 3000];

consumption =
  [| 1.5, 2.0, 1.5, 0.5, 0.1
   | 1.0, 0.0, 0.5, 1.0, 2.5
   | 1.0, 2.0, 1.0, 0.9, 0.1
   | 1.0, 0.0, 1.0, 1.5, 2.5 |];

profit = [11, 18, 15, 17, 11];
```

Production Planning: Solve



Production Planning: Other examples

- Toy shop planning problem 1:

availability of each type is a resource

each robot requires exactly one of the availability resources

```
PRODUCT = { A, B, C, D };
RESOURCE = { MONEY, AVAIL_A, AVAIL_B,
             AVAIL_C, AVAIL_D };
capacity = [ MONEY: 10000, AVAIL_A: 1000, AVAIL_B: 400,
             AVAIL_C: 500, AVAIL_D: 150];
consumption =
  [|          A:    B:    C:
   | MONEY:   13.0, 21.0, 17.0
   | AVAIL_A:  1.0,  0.0,  0.0
   | AVAIL_B:  0.0,  1.0,  0.0,   0.0
   | AVAIL_C:  0.0,  0.0,  1.0,   0.0
   | AVAIL_D:  0.0,  0.0,  0.0,   1.0 |];

profit = [A: 6.0, B: 10.0, C: 8.0, D: 40.0];
```

Production Planning: Other examples

- Toy shop planning problem 2:

```
PRODUCT = { KITT, HAL, T2000, GORT, R2 };
RESOURCE = { MONEY, AVAIL_KITT, AVAIL_HAL,
             AVAIL_T2000, AVAIL_GORT, AVAIL_R2 };
capacity = [10000, 3, 1, 9, 7, 10];
consumption =
  [| 400.0, 1600.0, 300.0, 500.0, 670.0
   | 1.0, 0.0, 0.0, 0.0, 0.0
   | 0.0, 1.0, 0.0, 0.0, 0.0
   | 0.0, 0.0, 1.0, 0.0, 0.0
   | 0.0, 0.0, 0.0, 1.0, 0.0
   | 0.0, 0.0, 0.0, 0.0, 1.0 |];
profit = [4.0, 40.0, 8.0, 12.0, 5.0];
```

Which solver should you use?

- We could use Gecode (the default solver)
  - it supports float variables and linear constraints
  - but solving takes a very long time (and finds lots of intermediate solutions before the optimal one)
- Better use Highs: it solves almost instantly!
- Production planning is a **M**ixed **I**nteger Linear **P**rogramming Problem, therefore **MILP** solvers like COIN-BC are ideal
  - constraints and objective are linear
  - variables are mix of integer and float

Summary

- Real models apply to data of different sizes
- MiniZinc uses
  - enumerated types to name objects
  - arrays to capture information about objects
  - array literals can be indexed
- Generic models (using arrays) allow us to reuse models for many instances
- Production planning is a MIP problem, so MIP solvers work best

Array Comprehensions

Peter Stuckey



MONASH
University

Array Comprehensions

- Comprehensions are used to construct arrays
- Known from many other programming languages
- Python: `[ 2 * x for x in range(20) if x*x > 3 ]`
- Haskell: `[ 2 * x | x <- [0..19], x^2 > 3 ]`
- MiniZinc: `[ x * 2 | x in 0..19 where x^2 > 3 ]`
- General form in MiniZinc:
  `[ expr | generator`$_1$ `where test`$_1$`, ...`
  `generator`$_k$ `where test`$_k$ `]`
- Each `generator` is of the form
  `identifier,...,identifier in expression`

Array Comprehensions: Examples

```
   [i + j | i, j in 1..4 where i < j]
= [1+2, 1+3, 1+4, 2+3, 2+4, 3+4]
= [3, 4, 5, 5, 6, 7]
```

```
   [ x[i] ≠ x[j] | i,j in 1..5 where i<j ]
= [ x[1]≠x[2], x[1]≠x[3], x[1]≠x[4],x[1]≠x[5],
    x[2]≠x[3], x[2]≠x[4], x[2]≠x[5],
    x[3]≠x[4], x[3]≠x[5],
    x[4]≠x[5] ]
```

```
   [ x[j,i] | i in 1..n, j in 1..m ]
= [ x[1,1], x[2,1], x[3,1], ..., x[m,1],
    ...
    x[1,n], x[2,n], x[3,n], ..., x[m,n] ]
```

Generators revisited

- Remember generator expressions?

```
sum(r in ROBOT) (cost[r]*buy[r]) <= budget;
```

- This is just special syntax for

```
sum([ cost[r]*buy[r] | r in ROBOT ]) <= budget
```

- It can be used with any function that takes a list as its argument, e.g.
  - for lists of numbers: `sum, product, min, max`
  - for lists of constraints: `forall, exists`

Example: pairwise different variables

- Constrain all pairs of variables in an array to take pairwise different values
- We will see later that this is one of the most useful constraints, at the core of many problems
- Easy to express with comprehensions/generators:

```
constraint
  forall (i,j in 1..n where i < j)
         (x[i] != x[j]);
```

Summary

- MiniZinc uses comprehensions extensively
  - to express constraints
  - to construct and transform arrays
- Only way to build arrays where the size is a parameter
- Generator calls are just syntactic sugar for comprehensions
- Indexed comprehensions create multi-dimensional arrays with specified index sets

Task Assignment

Peter Stuckey

MONASH
University

Staff Allocation

- This is a (very simple) version of a staff rostering problem.
  - We have a set of staff members and a number of shifts.
  - Each staff has a certain preference for each available shift (higher = better).
  - Each staff member can only work one shift in the roster.
  - Goal: maximise the overall preferences.

Staff Allocation: Data and Decisions

```
enum Staff;
enum Shift;

% Preferences (higher = better)
array[Shift,Staff] of int: preference;

% Decisions: who works which shift
array[Shift] of var Staff: allocation;
```

Staff Allocation: Constraint

- Assume we have 4 shifts, S1 to S4.
- No two shifts are assigned to the same staff:

```
constraint allocation[S1] ≠ allocation[S2];
constraint allocation[S1] ≠ allocation[S3];
constraint allocation[S1] ≠ allocation[S4];
constraint allocation[S2] ≠ allocation[S3];
constraint allocation[S2] ≠ allocation[S4];
constraint allocation[S3] ≠ allocation[S4];
```

- This is tedious to write, and depends on the data.
- How can we do this generically (if we don't know the number of shifts)?

Staff Allocation: Constraint

- We can of course use `forall` with a generator:

```
constraint
  forall (s1, s2 in Shift where s1 < s2)
         (allocation[s1] ≠ allocation[s2]);
```

- This is a very common pattern!
- The core of many assignment problems.
- Use a **global constraint** instead:

include functionality from the MiniZinc library

```
include "all_different.mzn";
constraint all_different(allocation);
```

enforces pairwise not-equals of an array

Global Constraints

- Historically: any constraint over **more than two variables**
  - So, technically, linear constraints are global
- In our context:
  - constraints that capture **common modelling patterns**
  - pre-defined in the **MiniZinc library**
- Global constraints make
  - models more **concise**
  - solving **faster** (MiniZinc and backend solvers can use information about the structure of the problem)

Staff Allocation: Objective

```
enum Staff;
enum Shift;

% Preferences (higher = better)
array[Shift,Staff] of int: preference;

% Decisions: who works which shift
array[Shift] of var Staff: allocation;

include "all_different.mzn";
constraint all_different(allocation);

solve minimize
    sum(s in Shift)
       (preference[s,allocation[s]]);
```

variable array access!

Variable Array Access

- We can use decision variables to index into arrays.
- This is a very powerful modelling capability!
- You've just seen a simple example:

  map the allocated staff (a decision variable) to their preferences
- We will make use of this a lot in future models.

Include *(not that important...)*

- Syntax:
  `include "filename";`
  - Includes the specified file in your model
  - Will search the MiniZinc library path, the path of the model file and the working directory
- Includes are used to
  - break up large models into more manageable chunks
  - load definitions of global constraints from the library
  - control how the model is translated for a particular solver (more on that in a later module!)

Summary

- Global constraints are one of the most important concepts in this course.
- They capture common model structure and make models more concise and efficient to solve.
- Using decision variables as array indexes is a powerful modelling tool.

Exercise: Cryptarithms

- A **cryptarithm** is a puzzle like this:

```
   T H I S
+      I S
= E A S Y
```

- All letters need to be assigned to **different** numbers 0-9
- The leading letters may not be 0
- Try the models from the browser:
  - https://tinyurl.com/isaac-cryptarithms

- smm: SEND+MORE=MONEY
- snake: SNAKE+SNAKE = RATTLE
- most: SEND+MOST=MONEY, optimising for maximum MONEY
- attempt: PLEASE+SOLVE+PUZZLE = QQQQQQ

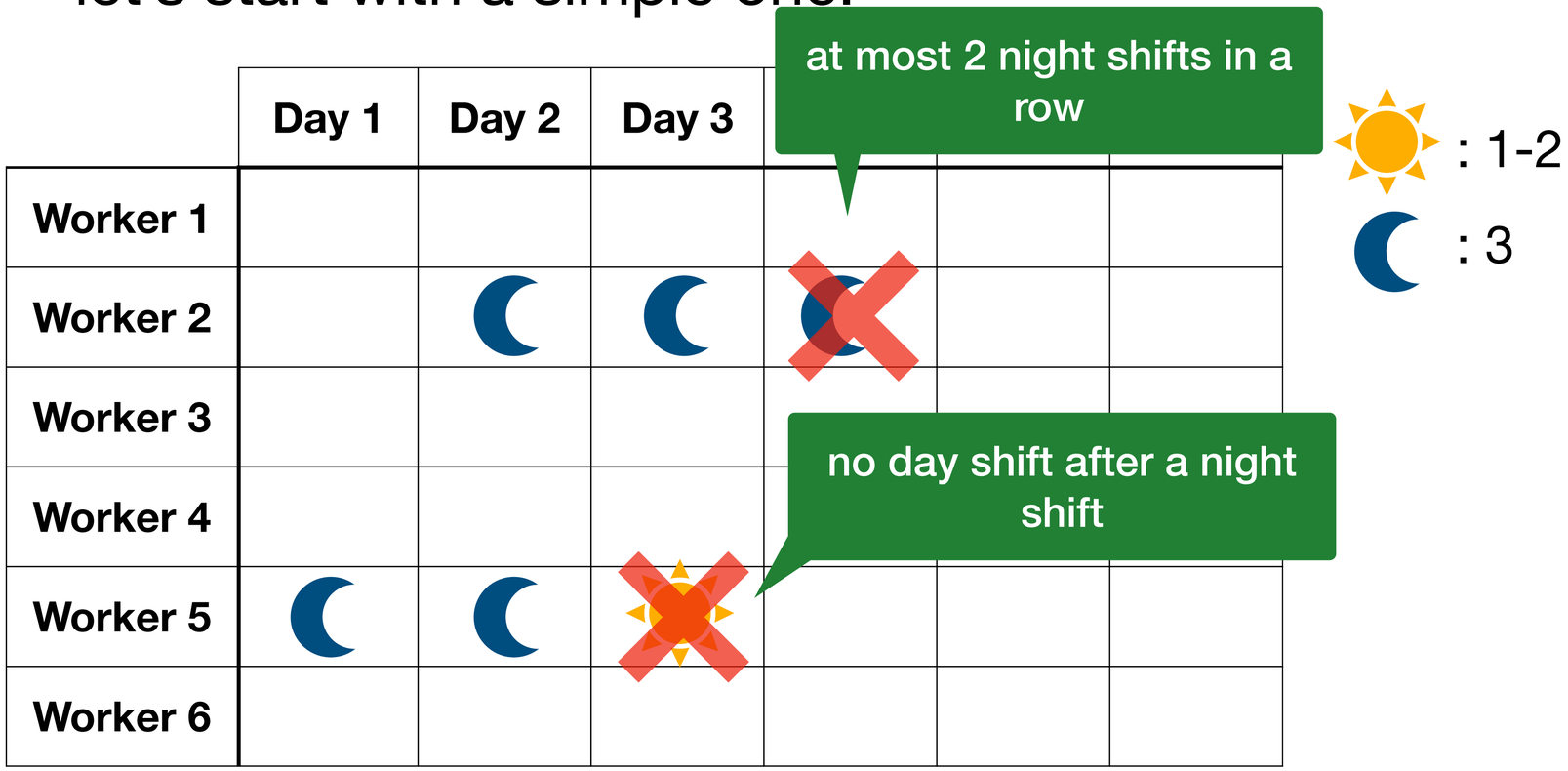
Staff Rostering

Peter Stuckey

MONASH
University

Staff Rostering

- Producing staff rosters is a common task in many industries.
- We will see more complicated versions later on, but let's start with a simple one:



Staff Rostering

- Here are the rules again:
  - For the next period of *n* days, we have to assign each staff member to either the day shift, the night shift, or no shift at all.
  - We have min/max requirements of staff per shift.
  - Nobody gets more than two night shifts in a row.
  - Nobody gets a day shift after a night shift.
  - Maximise the number of day shift staff.

Staff Roster as a Function

We can choose from several functional representations for this staff roster, mapping

- each **staff member** to a **set of shift types and days**:
  f: `Staff` → $2^{\texttt{Shift x Day}}$
- each **shift type** to a **set of staff members and days**:
  f: `Shift` → $2^{\texttt{Staff x Day}}$
- each **shift type and day** to a **set of staff members**:
  f: `Shift x Day` → $2^{\texttt{Staff}}$
- each **staff member and day** to a **shift type**:
  f: `Staff x Day` → `Shift`
- …

Partitioning Problems

- Finding a function f: **DOM** ➔ **COD**

  can be considered a **partitioning problem**

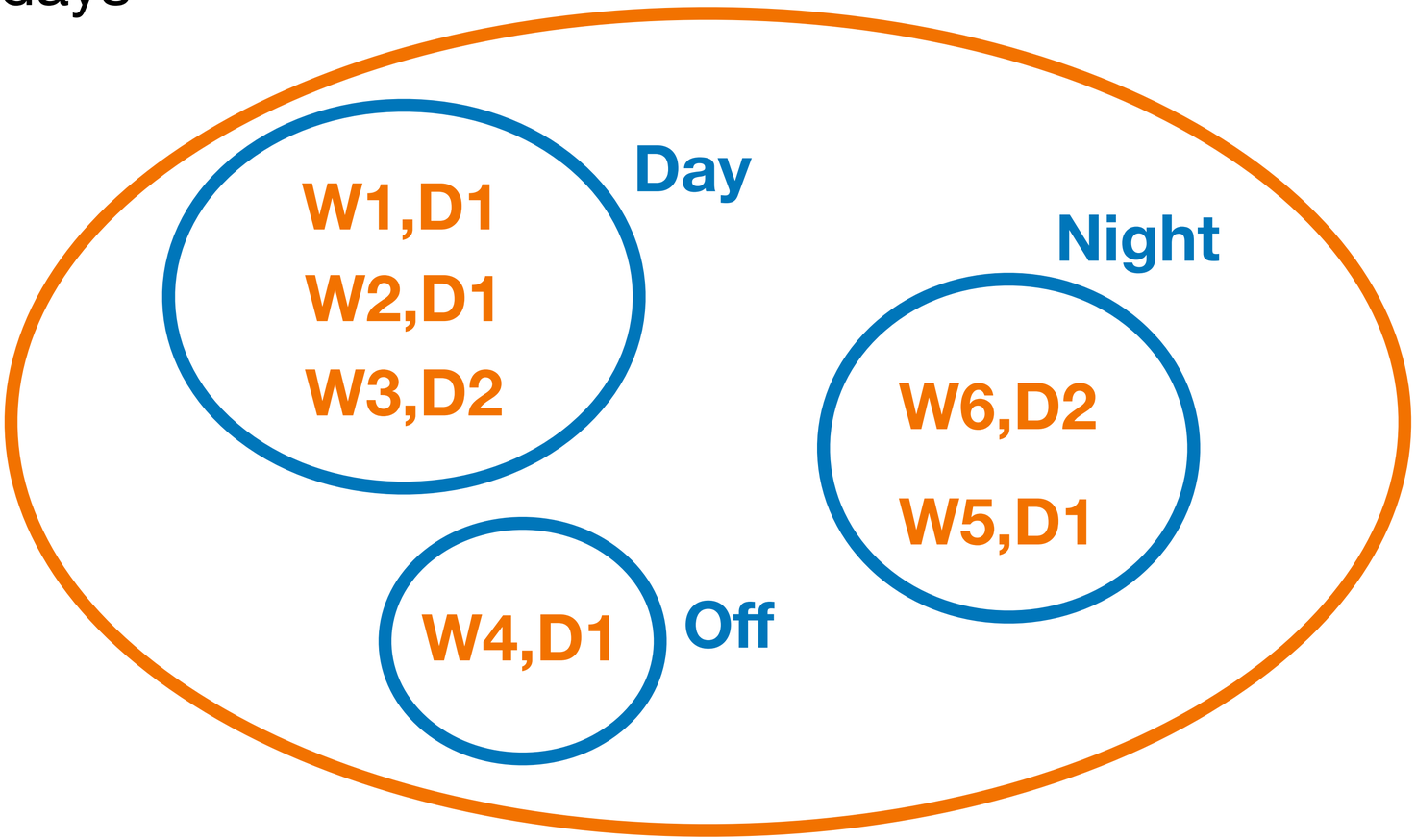
- This may give additional insight if
  we want to constrain or manipulate the sets

  F(c) = { d in **DOM** | f(d) = c },

  where F: **COD** ➔ 2^{DOM} *(pseudo-inverse of f)*

Staff Roster as a Partitioning Problem

- f maps each staff member and day to a shift type
- F maps each shift type to a set of staff members and days



Rostering Data

```
enum Staff;
enum Shift = { Day, Night, Off };
int: nDays;
set of int: WorkDay = 1..nDays;

int: reqNight; % required staff on night shift
int: l;        % lower bound for day shift
int: u;        % upper bound for day shift
```

Rostering Decisions

- **DOM** = `Staff` x `WorkDay`
- **COD** = `Shift` (one of `Day`, `Night` and `Off`)
- For each staff member and day, choose the shift:
  `array[Staff, WorkDay] of var Shift: roster;`
- No constraints required to ensure `roster` is a partition!
  - Staff can only work a single shift per day
  - Reading roster backwards, the shift types partition the set of combinations of staff and day

Rostering Pattern Constraints

"Nobody can have more than two night shifts in a row"

- How to express this?

```
forall (s in Staff, k in 1..nDays-2)
  (sum(d in k..k+2)(roster[s,d] = Night) <= 2)
```

- Hm, not very clear!
- Use logical connectives to be explicit:

```
forall (s in Staff, d in 1..nDays-2)
  (roster[s,d]=Night /\ roster[s,d+1]=Night
    -> roster[s,d+2] != Night);
```

- Same for night/day shift constraint:

```
forall (s in Staff, d in 1..nDays-1)
  (roster[s,d]=Night -> roster[s,d+1] != Day);
```

Logical Connectives

- Boolean expressions:
  - `true`
  - `false`
  - `/\` conjunction (“and”)
  - `\/` disjunction (“or”)
  - `->` implication (“if-then”)
  - `<->` bi-implication (“if-and-only-if”)
  - `not` negation
- Allow us to **combine constraints** in powerful ways
- But can be inefficient when **mixing logical and global constraints**… More later!

Shift Requirement Constraints

"There are `reqNight` staff members on night shift each day"

```
forall (d in WorkDay)
  (sum(s in Staff) (roster[s,d]=Night)
   = reqNight);
```

"There are between `l` and `u` people on each day shift:"

```
forall (d in WorkDay)
  (sum (s in Staff) (roster[s,d]=Day) >= l);

forall (d in WorkDay)
  (sum (s in Staff) (roster[s,d]=Day) <= u);
```

Objective

“As many staff members should be on day shifts as possible”

```
var int: totalDay =
  sum (d in WorkDay, s in Staff)
      (roster[s,d] = Day);

solve maximize totalDay;
```

Running the model

roster.mzn | roster.dzn

```
1  enum Staff;
2  enum Shift = { Day, Night, Off };
3  int: nDays;
4  set of int: Roster = 1..nDays;
5
6  int: reqNight; % required staff on night shift
7  int: l;        % lower bound for day shift
8  int: u;        % upper bound for day shift
9
10 array[Staff,Roster] of var Shift : roster;
11
12 constraint forall (s in Staff, d in 1..nDays-2)
13   (roster[s,d] = Night /\ roster[s,d+1] = Night -> roster[s,d+2] != Night);
14
15 constraint forall (s in Staff, d in 1..nDays-1)
16   (roster[s,d] = Night -> roster[s,d+1] != Day );
```

Output

Line: 9,Col: 1 2s 913msec

Running the model

- Solving… solving… solving…
- Two solutions are found quickly, but nothing more even after waiting several minutes.
- When changing the number of days from 8 to 6, the model can find an optimal solution after a few seconds.
- **What went wrong?**

Staff Rostering Again

Peter Stuckey

MONASH
University

Shift Requirement Constraints

"There are `reqNight` staff members on night shift each day"

```
forall (d in WorkDay)
  (sum(s in Staff) (roster[s,d]=Night)
   = reqNight);
```

"There are between `l` and `u` people on each day shift:"

```
forall (d in WorkDay)
  (sum (s in Staff) (roster[s,d]=Day) >= l);

forall (d in WorkDay)
  (sum (s in Staff) (roster[s,d]=Day) <= u);
```

common subexpressions

Auxiliary Variables

- Auxiliary variables
  - store values of expressions that are reused
  - are dependent on decisions
  - (note: auxiliary parameters too!)
- ```
  array[WorkDay] of var int: onDay =
     [ sum (s in Staff) (roster[s,d] = Day)
     | d in WorkDay ];
  constraint forall (d in WorkDay)
     (onDay[d] >= l /\ onDay[d] <= u);
  ```
- Or simply

  ```
  array[WorkDay] of var l..u: onDay =
    [ sum (s in Staff) (roster[s,d] = Day)
    | d in WorkDay ];
  ```
  ```

More Common Subexpressions

"As many staff members should be on day shifts as possible"

```
var int: totalDay =
  sum (d in WorkDay, s in Staff)
      (roster[s,d] = Day);

solve maximize totalDay;
```

Can be changed to

```
var int: totalDay = sum (onDay);
```

Improved Model

```
enum Staff;
enum Shift = { Day, Night, Off };
int: nDays;
set of int: WorkDay = 1..nDays;

int: reqNight; % required staff on night shift
int: l;        % lower bound for day shift
int: u;        % upper bound for day shift

array[Staff,WorkDay] of var Shift : roster;

constraint forall (s in Staff, d in 1..nDays-2)
  (roster[s,d] = Night /\ roster[s,d+1] = Night ->
roster[s,d+2] != Night);

constraint forall (s in Staff, d in 1..nDays-1)
  (roster[s,d] = Night -> roster[s,d+1] != Day );
```

Improved Model (continued)

```
array[WorkDay] of var l..u: onDay =
  [ sum (s in Staff) (roster[s,d] = Day)
  | d in WorkDay ];

constraint forall (d in WorkDay)
  (sum (s in Staff) (roster[s,d] = Night)
   = reqNight);

var int: totalDay = sum (onDay);
solve maximize totalDay;
```

Running the Improved Model

Output

```
Running roster-cse.mzn with roster.dzn
[| S1: Night, Night,   Off, Night, Night,   Off, Night, Night |
   S2: Night, Night,   Off, Night,   Off, Night, Night,   Off |
   S3:   Off,   Day, Night, Night,   Off,   Day, Night, Night |
   S4: Night,   Off, Night,   Off, Night, Night,   Off, Night |
   S5:   Day, Night, Night,   Off, Night, Night,   Off,   Day |
   S6:   Day,   Day,   Day,   Day,   Day,   Day,   Day,   Day |]
----------
[| S1: Night, Night,   Off, Night, Night,   Off, Night, Night |
   S2: Night,   Off, Night, Night,   Off, Night, Night,   Off |
   S3:   Off,   Day, Night, Night,   Off,   Day, Night, Night |
   S4: Night, Night,   Off,   Day, Night, Night,   Off, Night |
   S5:   Day, Night, Night,   Off, Night, Night,   Off,   Day |
   S6:   Day,   Day,   Day,   Day,   Day,   Day,   Day,   Day |]
----------
==========
Finished in 3s 645msec
```

Partitioning Problems with Cardinalities

- Many partitioning problems require bounds on the size of the partitions

  (e.g. #Night = 3, 1 ≤ #Day ≤ 2)

- We have global constraints for partitioning with size bounds:

  `global_cardinality(x, v, c)`

  means `c[i] = sum(j in x) (j = v[i])`

- E.g. `x = [A,A,F,G,C], v = [A,B,C], c = [2,0,1]`

- It combines counting constraints for different values `v` (e.g. `Day`, `Night`, `Off`) into a single global constraint

Roster with Global Cardinality

Replace

```
array[WorkDay] of var 1..u: onDay =
  [ sum (s in Staff) (roster[s,d] = Day)
  | d in WorkDay ];

constraint forall (d in WorkDay)
  (sum (s in Staff) (roster[s,d] = Night)
   = reqNight);
```

with

```
array[WorkDay] of var 1..u: onDay;
constraint forall (d in WorkDay)
  (global_cardinality([roster[s,d] | s in Staff],
                      [Day,       Night],
                      [onDay[d], reqNight]));
```

For this model, about 2x faster than sum!

Global Cardinality Variants

The MiniZinc library has a number of variants of global cardinality.

- Collecting counts, and requiring no `x` takes values outside of `v`
  `global_cardinality_closed(x, v, c)`
- Bounding the count of the number of occurrences:
  `global_cardinality_low_up(x, v, lo, hi)`
- Bounding the count, and requiring that no `x` takes values outside of `v`:
  `global_cardinality_low_up_closed(x,v,l,u)`

Summary

- Many discrete optimization problems involve
  - deciding a function f: **DOM** → **COD**
  - this can be seen as **partitioning** DOM
- The partition view is useful when we reason about the sets F(c) (pseudo-inverse of f)
- Partitioning with cardinality constraints is a substructure captured by the `global_cardinality` family
- Common subexpressions
- Logical connectives

Viewpoints

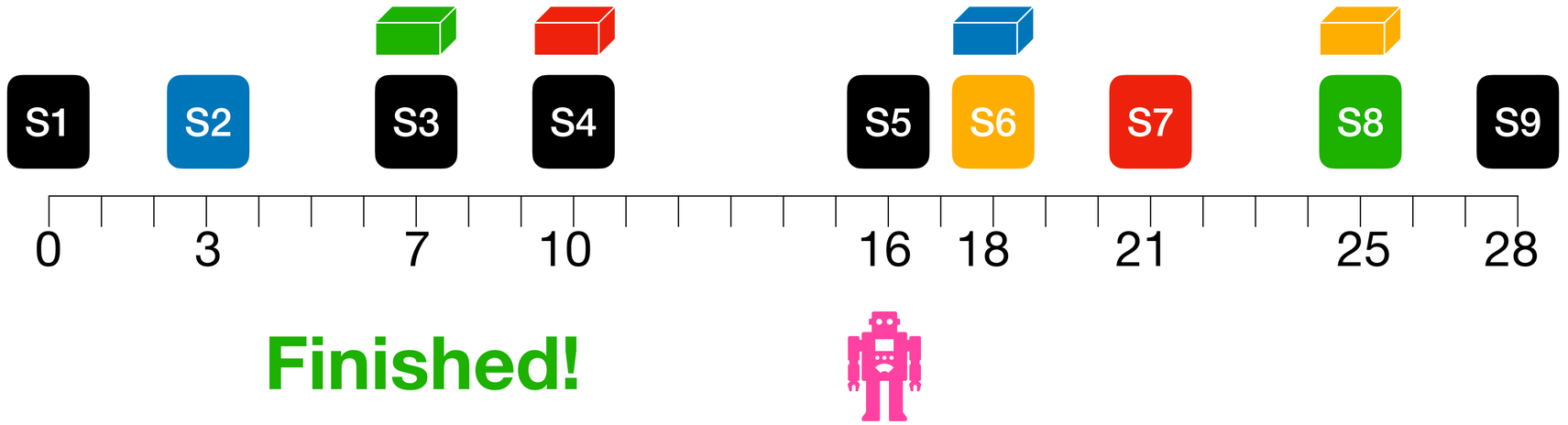
Peter Stuckey



MONASH
University

Robot Pick-up and Drop-Off

- We have a robot that moves along a linear rail. It needs to visit a set of work stations.



- Some work stations have items that must be taken to a given target work station. Other stations just need to be visited.

- The goal is to do this as quickly as possible, which means to minimise the distance the robot is travelling.

TSP on a Line

- This is an instance of the *Travelling Salesperson Problem (TSP)*
- In the general TSP, points are not located on a line but generally in 2D space
- In addition to normal TSPs, we have *precedence constraints* between certain pairs of locations.

Permutation Problem

- This can again be regarded as a **permutation problem**: place a set of objects **OBJ** in an order
- We can regard it as a matching problem, matching the *n* objects in **OBJ** with numbers 1..*n*
- At least two different viewpoints:
  - **DOM** = **OBJ**, **COD** = 1..*n*
  - **DOM** = 1..*n*, **COD** = **OBJ**
- The robot TSP is a permutation problem.

Pick-up/Drop-off Model: Data

```
enum Workstation;
int: n = card(Workstation);
Workstation: first;

array[Workstation] of int: coord;

enum Packet;
array[Packet] of Workstation: pickup;
array[Packet] of Workstation: dropoff;

% Workstation = {W1,W2,W3,W4,W5,W6,W7,W8,W9};
% Packet = {blue, red, yellow, green};
% pickup  = [W6, W4, W8, W3];
% dropoff = [W2, W7, W6, W8];
```

Pick-up/Drop-off Model: Viewpoints

- We will use both viewpoints:
  - The position of each workstation in the order:
    ```
    set of int: Pos = 1..n;
    array[Workstation] of var Pos: order;
    ```
  - The workstation at each step in the route:
    ```
    array[Pos] of var Workstation: route;
    ```
- Constraints:

```
route[1] = first;

alldifferent(order);

forall (pk in Packet)
  (order[pickup[pk]] < order[dropoff[pk]]);
```

Channeling using inverse

- If we have two viewpoints of a problem they should agree

```
forall(w in WorkStation, p in Pos)
      (order[w] = p <-> route[p] = w)
```

- Since this is a common kind of channeling, the MiniZinc library has a global constraint for it:

```
inverse(order,route)
```

- It states that the function represented by array `order` is the inverse of that represented by `route` (and vice versa).

- It makes the `alldifferent` redundant

Pick-up/Drop-off Model: Objective

```
var int: totalDistance;
constraint totalDistance =
  sum (p in 1..n-1)
      (abs(coord[route[p]] - coord[route[p+1]]));

solve minimize totalDistance;
```

Running the Model

Output

```
totalDistance = 79;
----------
order = [9, 8, 2, 3, 1, 5, 6, 4, 7];
route = [W5, W3, W4, W8, W6, W7, W9, W2, W1];
totalDistance = 72;
----------
order = [9, 8, 2, 3, 1, 6, 7, 4, 5];
route = [W5, W3, W4, W8, W9, W6, W7, W2, W1];
totalDistance = 64;
----------
order = [9, 8, 2, 3, 1, 7, 6, 4, 5];
route = [W5, W3, W4, W8, W9, W7, W6, W2, W1];
totalDistance = 58;
----------
==========
Finished in 65msec
```

Summary

- The Pick-up/Drop-off problem is a simplification of the classic **Traveling Salesperson Problem** (TSP) in computer science, but our example has also side constraints (precedence)
- The TSP is an important problem in graph theory and has applications to routing and optimisation in general
- In our example, some requirements are impossible to express in a certain viewpoint, making the combined model the only way to formulate the complete problem

Scheduling

Peter Stuckey

MONASH
University

Basic Scheduling

- Scheduling is an important class of discrete optimisation problems
- Basic scheduling involves:
  - tasks with a **duration**
  - **precedences** between tasks: one task must complete before another can start
- The aim is to schedule the tasks and usually to minimize the latest end time

Modeling Time

- In discrete optimization time is modeled by integers (not continuous)
- Time variables tend to have VERY large ranges
  - e.g. start times on the minute for a 7 day schedule
- We typically only care about
  - earliest time, or
  - latest time
- when reasoning (not about all possible times)

Basic Scheduling Data & Decisions

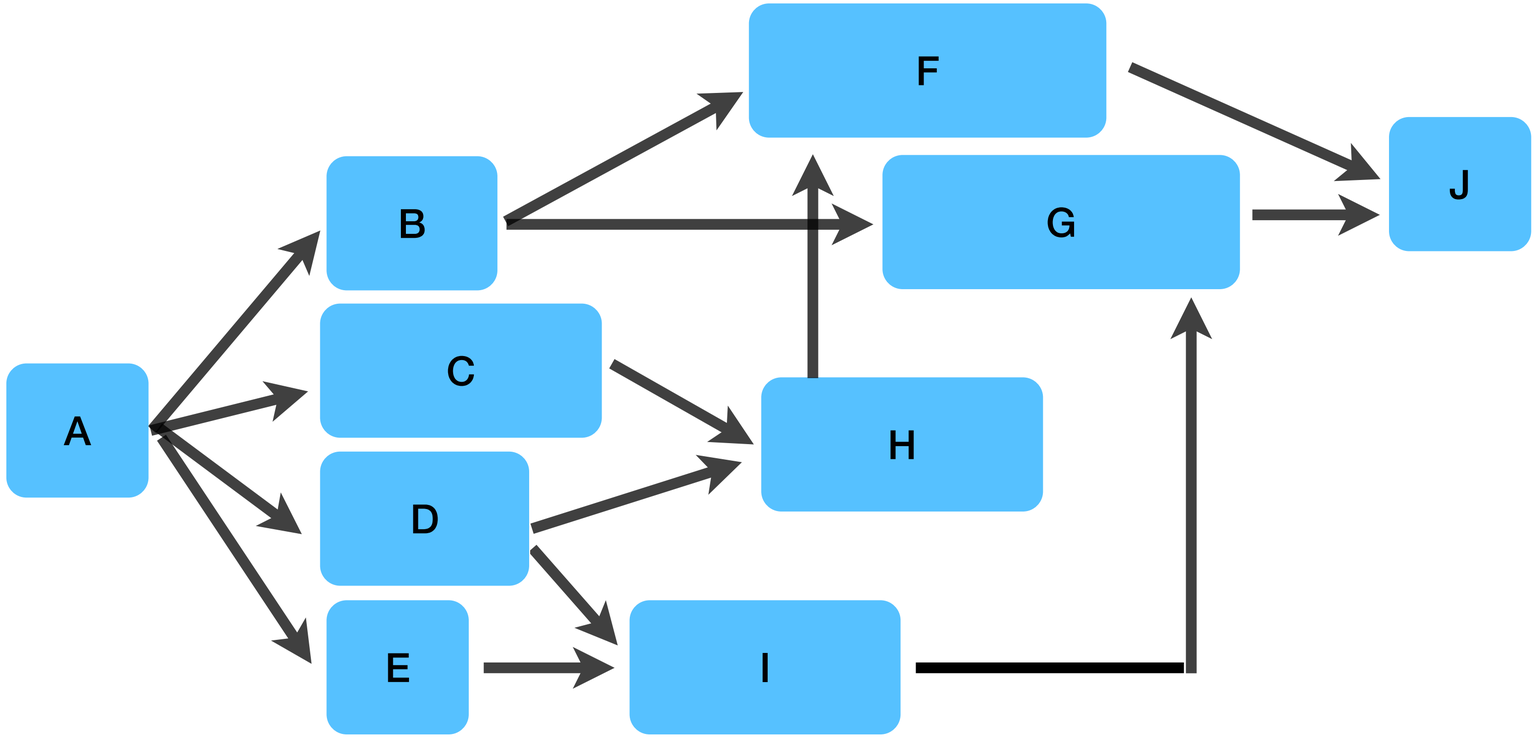
```
enum TASK;

array[TASK] of int: duration;
array[int] of tuple(TASK,TASK): pre;

int: e = sum(duration);
array[TASK] of var 0..e: start;
```

Tuple types
e.g. a pair of tasks

Basic Scheduling Instance



- Length indicates durations
- Arcs indicate precedences

Basic Scheduling Data & Decisions

```
TASK = {A, B, C, D, E, F, G, H, I, J};

duration = [10,10,20,15,10,25,20,20,20,10];

pre =
  [ (A, B),  (A, C),
    (A, D),  (A, E),
    (B, F),  (B, G),
    (F, J),  (G, J),
    (C, H),  (D, H),
    (D, I),  (E, I),
    (H, F),  (I, G) ];
```

Constraints & Objective

- We can force the precedences of the tasks

```
constraint forall(p in pre)
  (start[p.1]] + duration[p.1]]
     <= start[p.2]]);
```

Iterating over tuples in the array

t.1 is the first element of a tuple t

t.2 is the second element of a tuple t

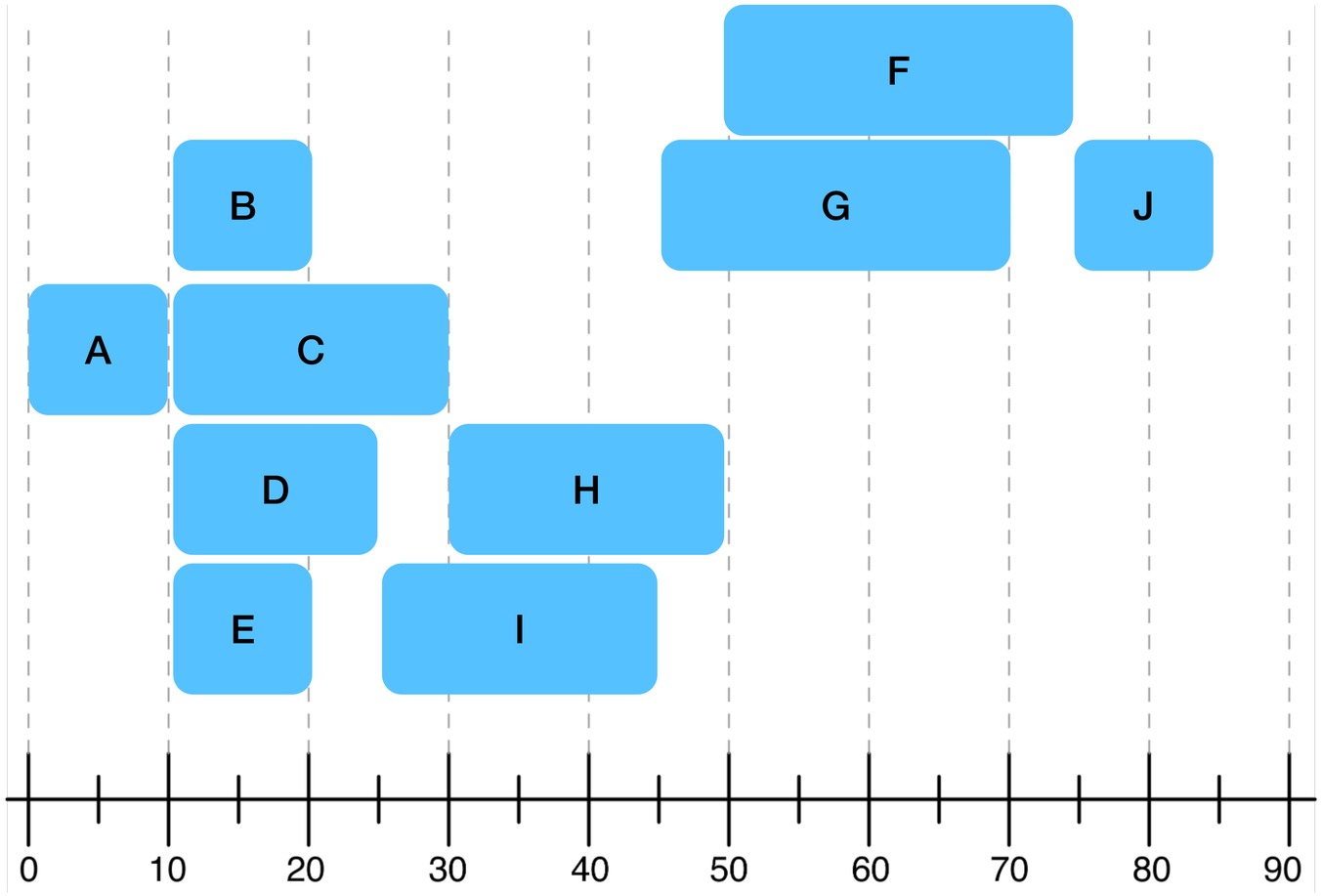
- To minimize the total time taken we use

```
var 0..e: makespan =
    max(t in TASK)(start[t] + duration[t]);
solve minimize makespan;
```

Constraints Generated

```
start[A] + 10 <= start[B]
start[A] + 10 <= start[C]
start[A] + 10 <= start[D]
start[A] + 10 <= start[E]
start[B] + 10 <= start[F]
start[B] + 10 <= start[G]
start[F] + 25 <= start[J]
start[G] + 20 <= start[J]
start[C] + 20 <= start[H]
start[D] + 15 <= start[H]
start[D] + 15 <= start[I]
start[E] + 10 <= start[I]
start[H] + 20 <= start[F]
start[I] + 20 <= start[J]
```

Basic Scheduling Solution




```
makespan     A   B  C   D   E   F   G   H   I   J
85         = [0, 10, 10, 10, 10, 50, 45, 30, 25, 75]
```

Difference Logic Constraints

- **Difference logic constraints** take the form `x + d ≤ y`, where `d` is constant

- Note $x + d = y \leftrightarrow x + d \leq y \wedge y + (-d) \leq x$

- A problem consists solely of difference logic constraints can be rapidly solved as a **longest/shortest path problem**

- But adding extra constraints means this advantage disappears

  - e.g., at most two tasks can run simultaneously

Summary

- Basic scheduling problems are a common part of many complex discrete optimisation problems
  - tasks with precedences
- The constraints needed to model this are a **simple form** of linear constraints: **difference logic constraints**
- Problems involving only these constraints can be solved **very efficiently**

Resources

- Critical to most scheduling problems are **limited resources**
- There are two types of resources
  - **Unary resources**: at most one task at a time
  - **Cumulative resources**: a limit on the amount of resource used at any time

Unary Resources

- The Project Scheduling problem with non-overlap involved a unary resource
  - number of tasks executing at one time
- Unary resources are common
  - machine
  - nurse, doctor, worker in a roster
  - track segment (one train at a time)

Scheduling Concepts (so far)

- Tasks
  - start time, duration (and end time)
  - other attributes

```
array[TASK] of var int: start;
array[TASK] of (var) int: duration;
```

- Precedences
  - one task can only start after another finishes
  - task t1 precedes t2

```
start[t1] + duration[t1] <= start[t2]
```

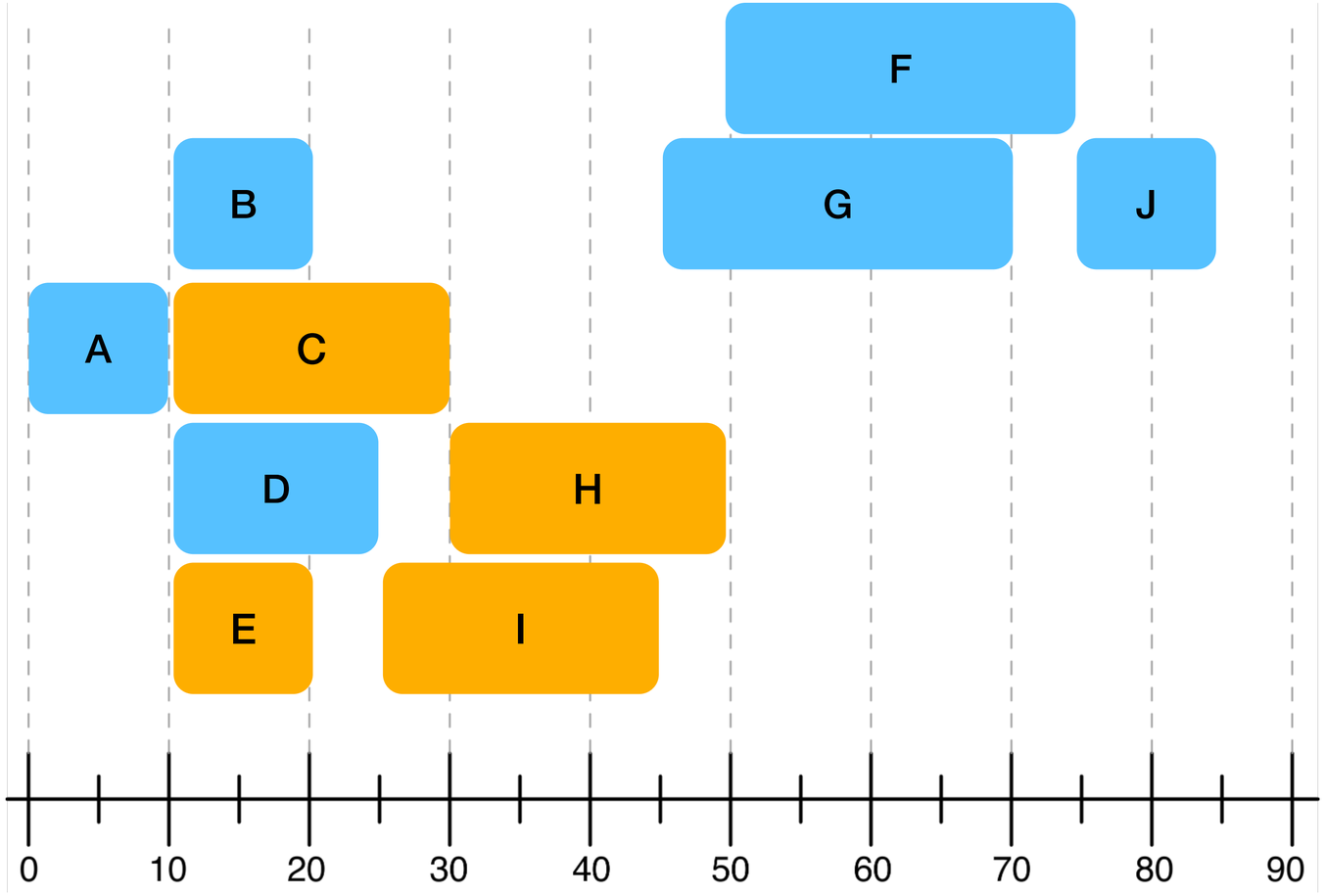
Nonoverlap

- Tasks that use the same unary resource cannot overlap

```
predicate nonoverlap(var int:s1, var int:d1,
                     var int:s2, var int:d2) =
          s1 + d1 <= s2 \/ s2 + d2 <= s1;

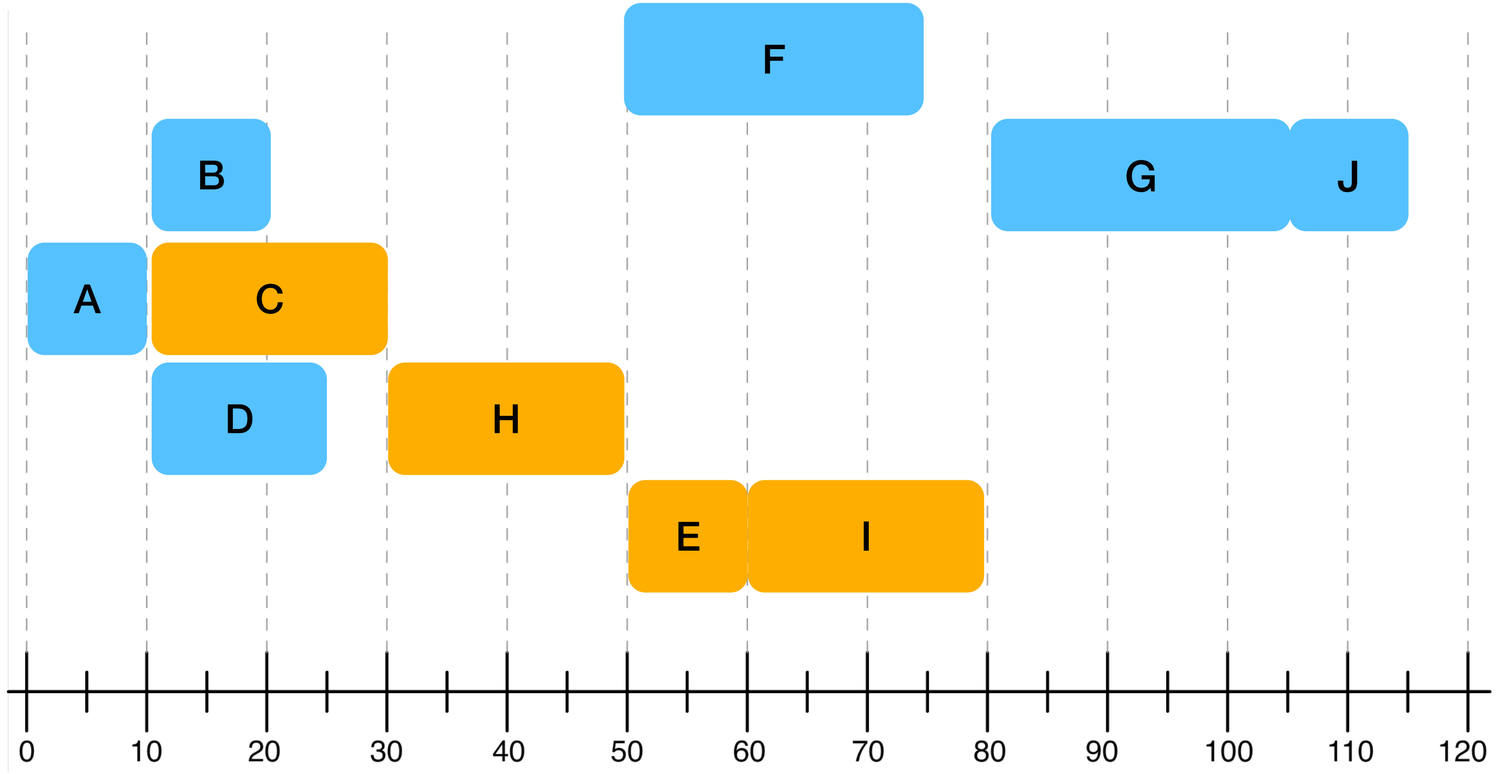
set of TASK: MUTEX = {C, E, H, I};
constraint forall(t1, t2 in MUTEX where t1 < t2)
      (nonoverlap(start[t1],duration[t1],
                  start[t2],duration[t2]));
```

Basic Scheduling Solution




```
makespan     A   B   C   D   E   F   G   H   I   J
85         = [0, 10, 10, 10, 10, 50, 45, 30, 25, 75]
```

Scheduling with Unary Resource




```
makespan     A   B   C   D   E   F   G   H   I   J
115        = [0, 10, 10, 10, 50, 50, 80, 30, 60, 105]
```

Disjunctive Global Constraint

- `nonoverlap` only considers two tasks at a time
  - unary resources require non overlap for all pairs of tasks that use it
- The **disjunctive constraint** ensures that no two tasks in the array overlap in execution

```
predicate disjunctive(array[int] of var int:s,
                      array[int] of var int:d)
=
   forall(i1,i2 in index_set(s) where i1 < i2)
        (nonoverlap(s[i1],d[i1],s[i2],d[i2]));
```

Replacing Nonoverlap with Disjunctive

- Tasks that use the same unary resource cannot overlap

```
include “disjunctive.mzn";

constraint disjunctive(
  [start[t] | t in MUTEX],
  [duration[t] | t in MUTEX],
);
```

Summary

- Disjunctive scheduling
  - allows us to express that two tasks do not overlap in execution without specifying the relative order
- Disjunctive global constraint
  - capture a set of tasks on a unary resource

Exercise: ExamStudy

- Generate a time table for exam study over a number of days
- Each topic needs a certain number of hours
- Some topics need to be completed before others begin
- Some topics must be started first thing in the morning
- Some days only the morning is available to study (sports commitments)
- [Extension] Each topic must be completed in a single day
- You should use **scheduling constraints**
- https://tinyurl.com/minizinc-examstudy

Cumulative Scheduling

Peter Stuckey

MONASH University

Resource Constraint Project Scheduling (RCPSP)

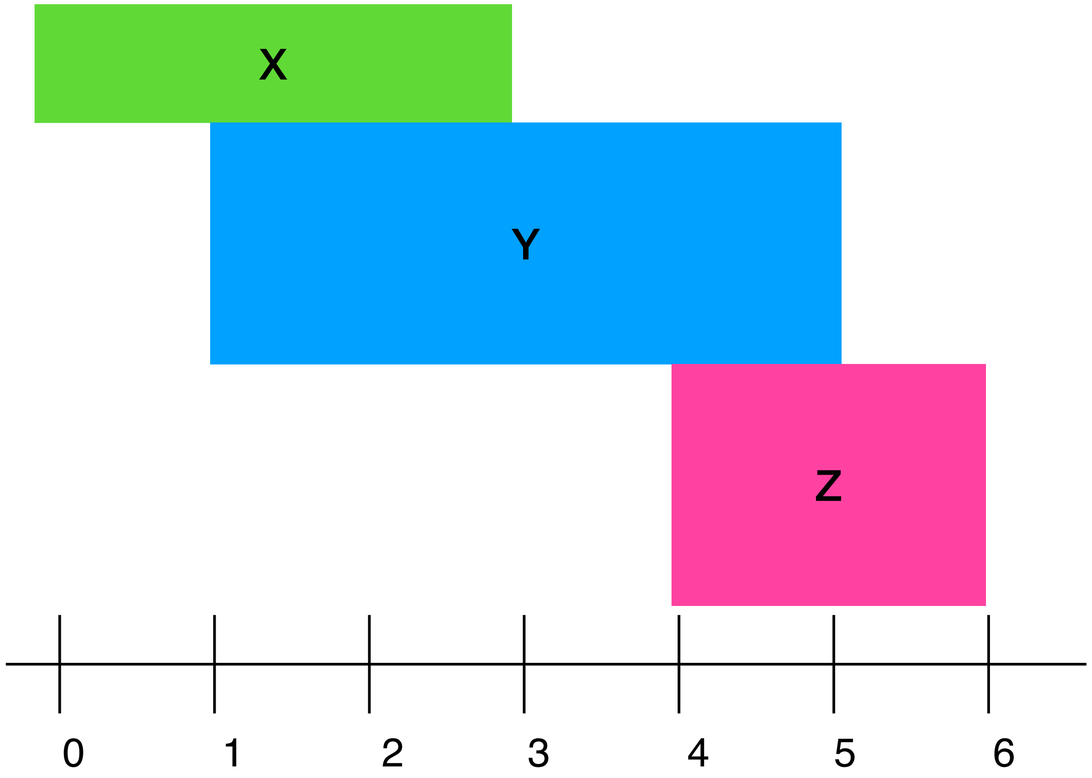
- Given tasks `t in TASK`
- Given precedences `p in PREC`
  - `pre[p,1]` precedes `pre[p,2]`
- Assume resources `r in RESOURCES`
- Each task `t` needs `res[r,t]` resources during its execution
- We have a limit `L[r]` for each resource
- Find the shortest schedule to run every task!
- Possibly the **most studied scheduling problem**

Resources

- Unary resources are unique
- Often we have **multiple identical copies** of a resource
  - bulldozers
  - workers (of equal capability)
  - operating theaters
  - airplane gates
- How do we model multiple identical resources?
  - assume task `t` uses `res[t]`
  - assume a limit `L` of resource **at all times**

Visualizing Resource Requirements

- A task `t` is a box of *length* `duration[t]` and *height* `res[t]`, starting at time `start[t]`



Modeling Resources: Time Decomposition

- The use of the resource at each time `i` is less that the limit `L`

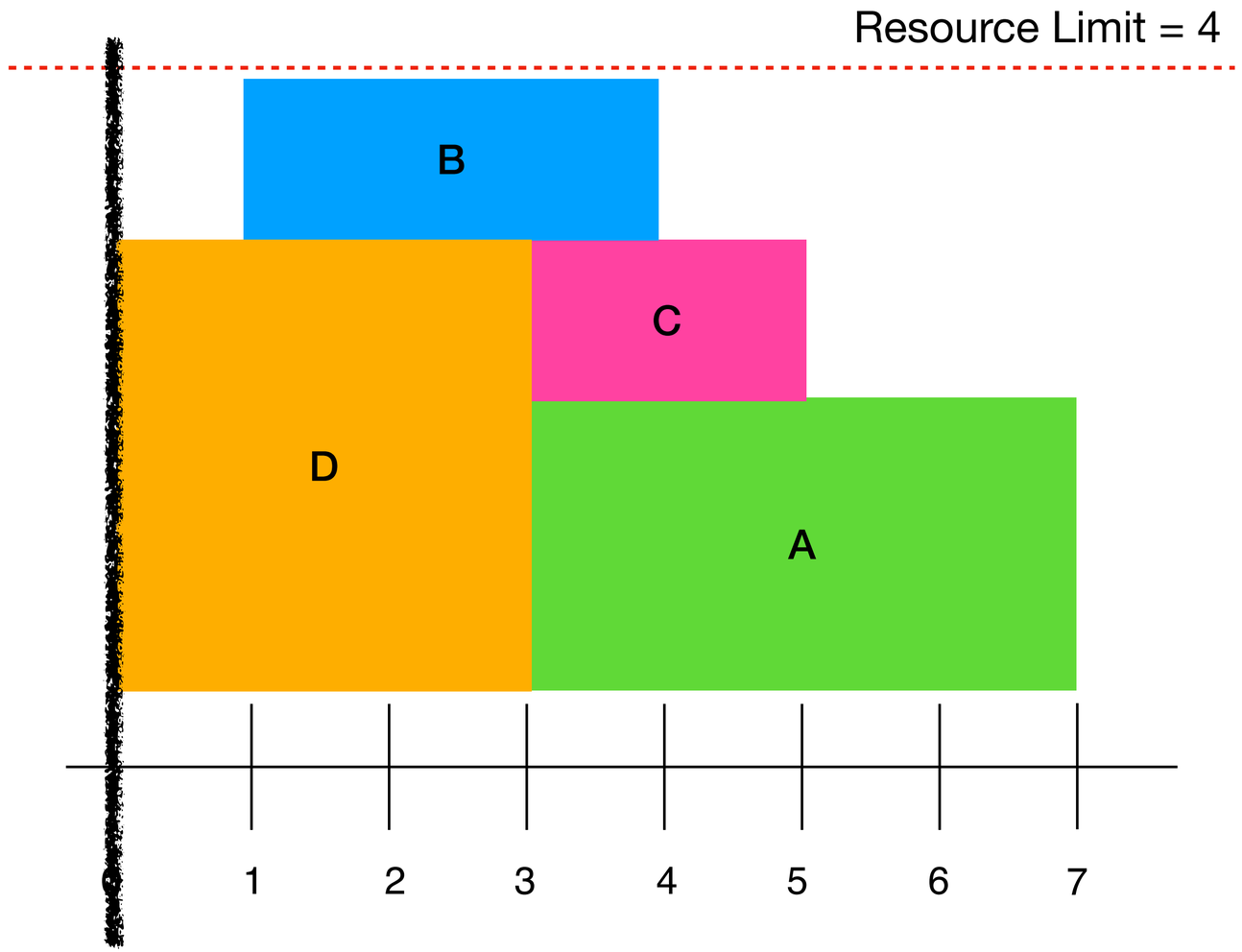
```
forall(i in TIME)(sum(t in TASK)(
  (start[t]<=i /\ start[t]+duration[t]>i)
   * res[t]
) <= L);
```

- Note the expression

```
start[t]<=i /\ start[t]+duration[t]>i
```

  represents whether task `t` runs at time `i`

Modeling Resources: Time Decomposition



Modeling Resources: Time Decomposition

- The use of the resource at each time `i` is less that the limit `L`

```
forall(i in TIME)(sum(t in TASK)
      ((start[t]<=i /\ start[t]+duration[t]>i)
        * res[t]) <= L);
```

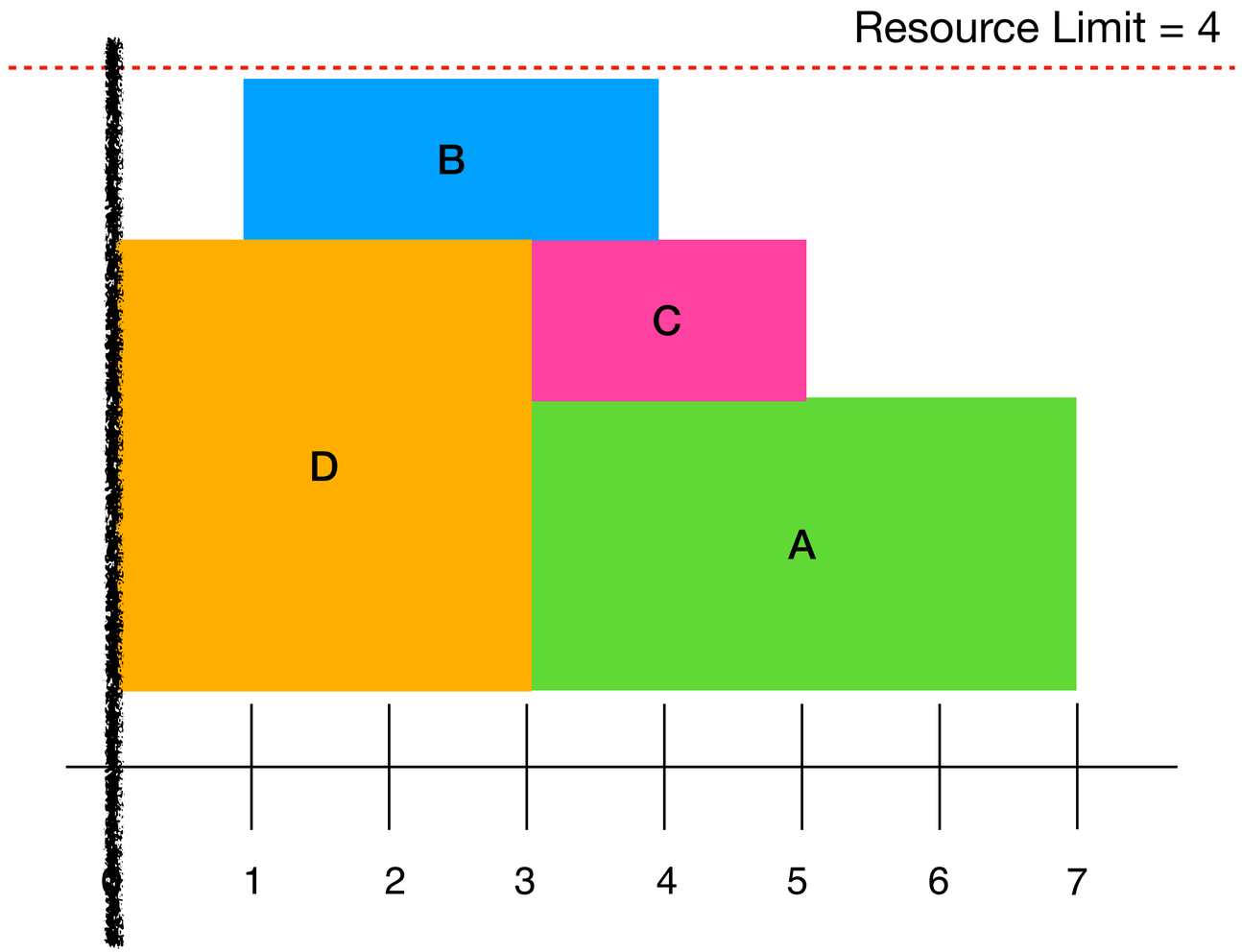
- **Problem**: size is `card(TASK)*card(TIME)`
  - many time periods in `TIME`

Modeling Resources: Task Decomposition

- Note we can overload a resource **only when** a task starts (otherwise no increase)
- Alternate model: only check start times

```
forall(t2 in TASK)(sum(t in TASK)(
   (start[t] <= start[t2] /\ start[t] +
    duration[t] > start[t2])
   * res[t]
)<= L);
```

Modeling Resources: Task Decomposition



Modeling Resources: Task Decomposition

- A more explicit formulation

```
forall(t2 in TASK)(sum(t in TASK where t != t2)
  ((start[t] <= start[t2] /\
    start[t] + duration[t] > start[t2])
   * res[t]) + res[t2] <= L);
```

- Comparison with time decomposition
  - **Advantage**: much smaller than time decomposition with `card(TASK)`^{2}
  - **Problem**: not as much information (fewer constraints) to the solver

Cumulative Global Constraint

- The **cumulative** global constraint captures exactly a resource constraint

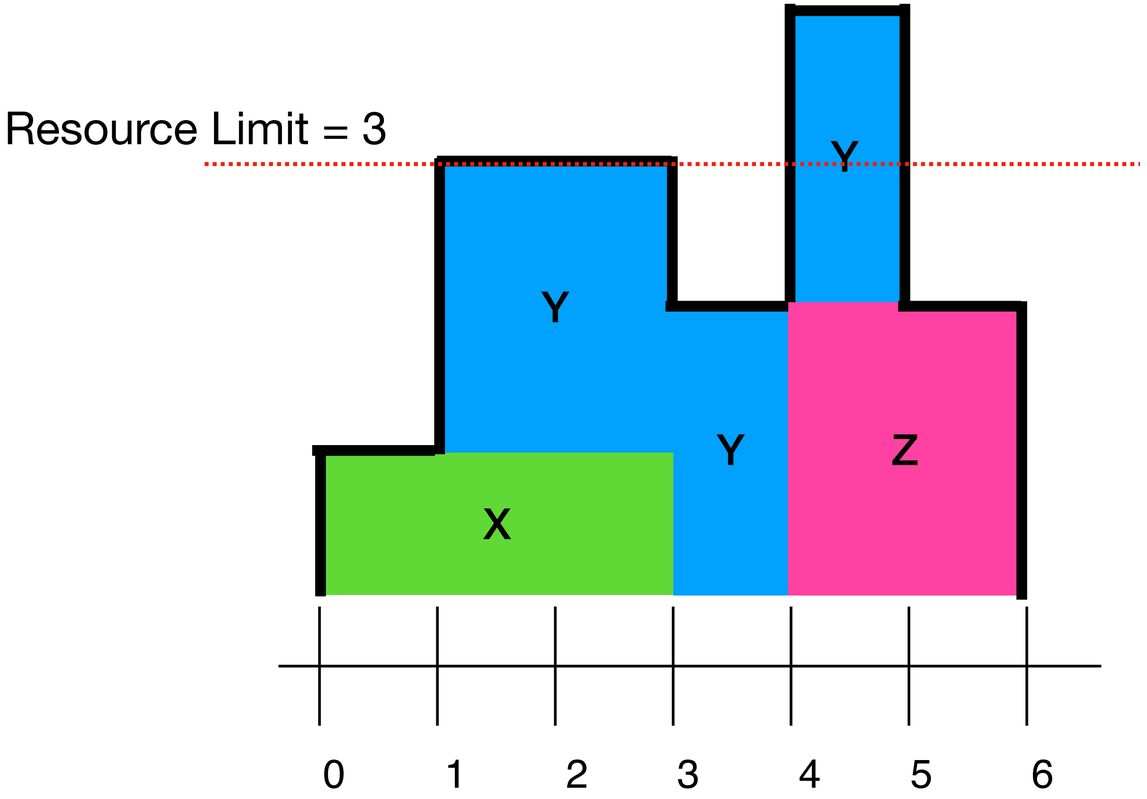
```
cumulative(<start time array>, <duration array>,
           <resource usage array>, <limit>)
```

  - ensure no more than the limit of the resource is used at any time during the execution of tasks

```
predicate cumulative(
  array[int] of var int: s,
  array[int] of var int: d,
  array[int] of var int: r,
  var int: L
);
```

Visualizing Cumulative

- They are not really boxes
- Timetable (black skyline) shows the usage



Summary

- There is a lot of research in how to propagate cumulative constraints
  - timetable propagation
    - equivalent to the time decomposition, but faster than the task decomposition
  - edge finding
    - reasoning about time intervals rather than single times
  - energy based reasoning
    - more inference than edge finding, but slower
  - TTEF time table edge finding
    - a combination of timetable with some energy based reasoning
    - state of the art

Summary

- Renewable capacitated resources
  - a resource capacity available over the schedule
- Time decomposition:
  - check resource usage at each time
- Task decomposition
  - check resource usage as each task starts
- `cumulative` global constraint

- RCPSP: a core scheduling problem

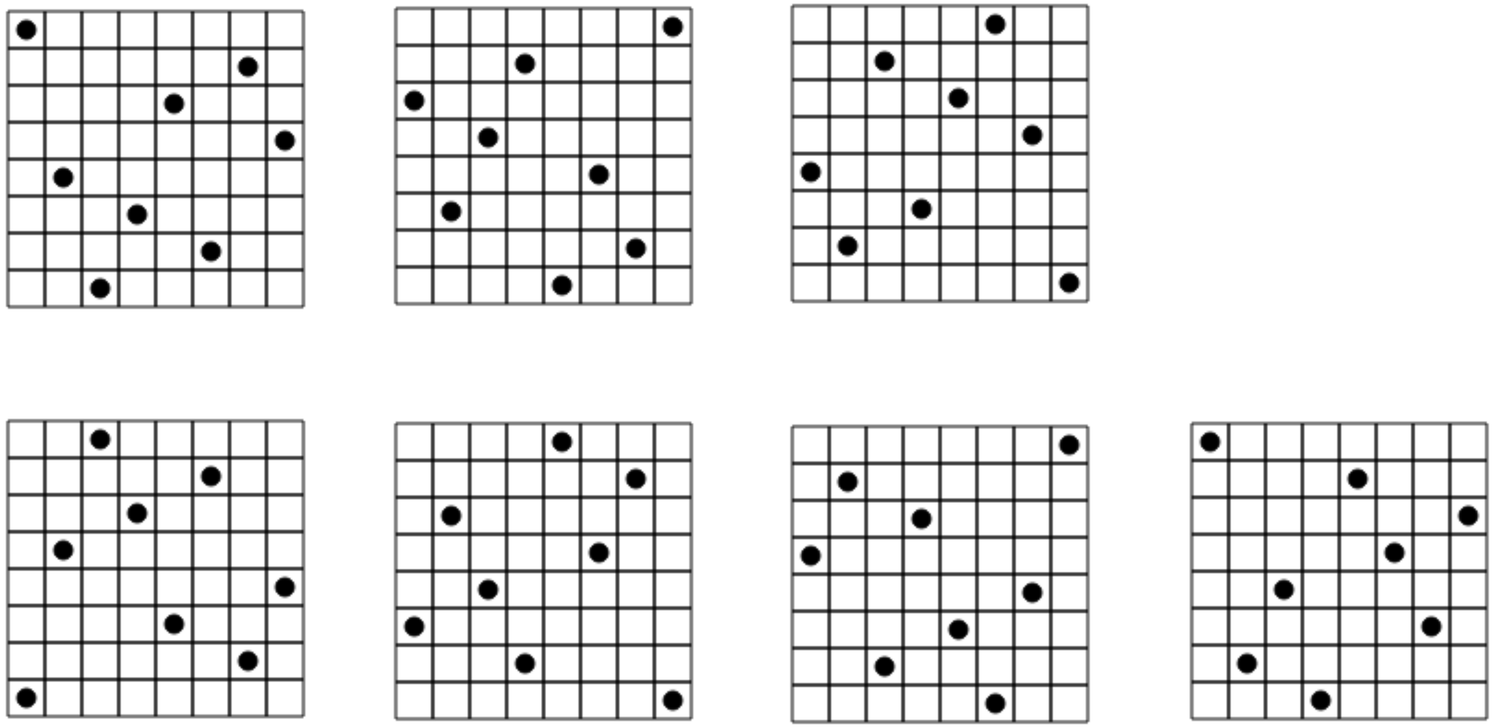
Advanced modelling

Modelling is like Programming

- You can write efficient and inefficient models
- The model is **declarative:**
  - it specified **what** the solution is, not how it is obtained (compared to an algorithm)
- The solving approach **is not declarative:**
  - you need to understand a bit about the solver to write efficient models

Symmetry Breaking

Symmetry in n-queens



Each solution has 7 symmetric variants.

Each **non-solution** has 7 symmetric variants, too!

Static Symmetry Breaking

- Add constraints to the model that only allow one solution per group of symmetric solutions
- **Q:** how could you eliminate horizontal reflection in 8-queens?
- In general: only keep **lexicographically smallest** solution
- If q^s is a symmetric permutation of `q`, add the constraint `lex_lesseq(q,`q^s`)`

Lexicographic Constraints

A	B	C
D	E	F
G	H	I

≤

C	B	A
FD	E	D
I	H	G

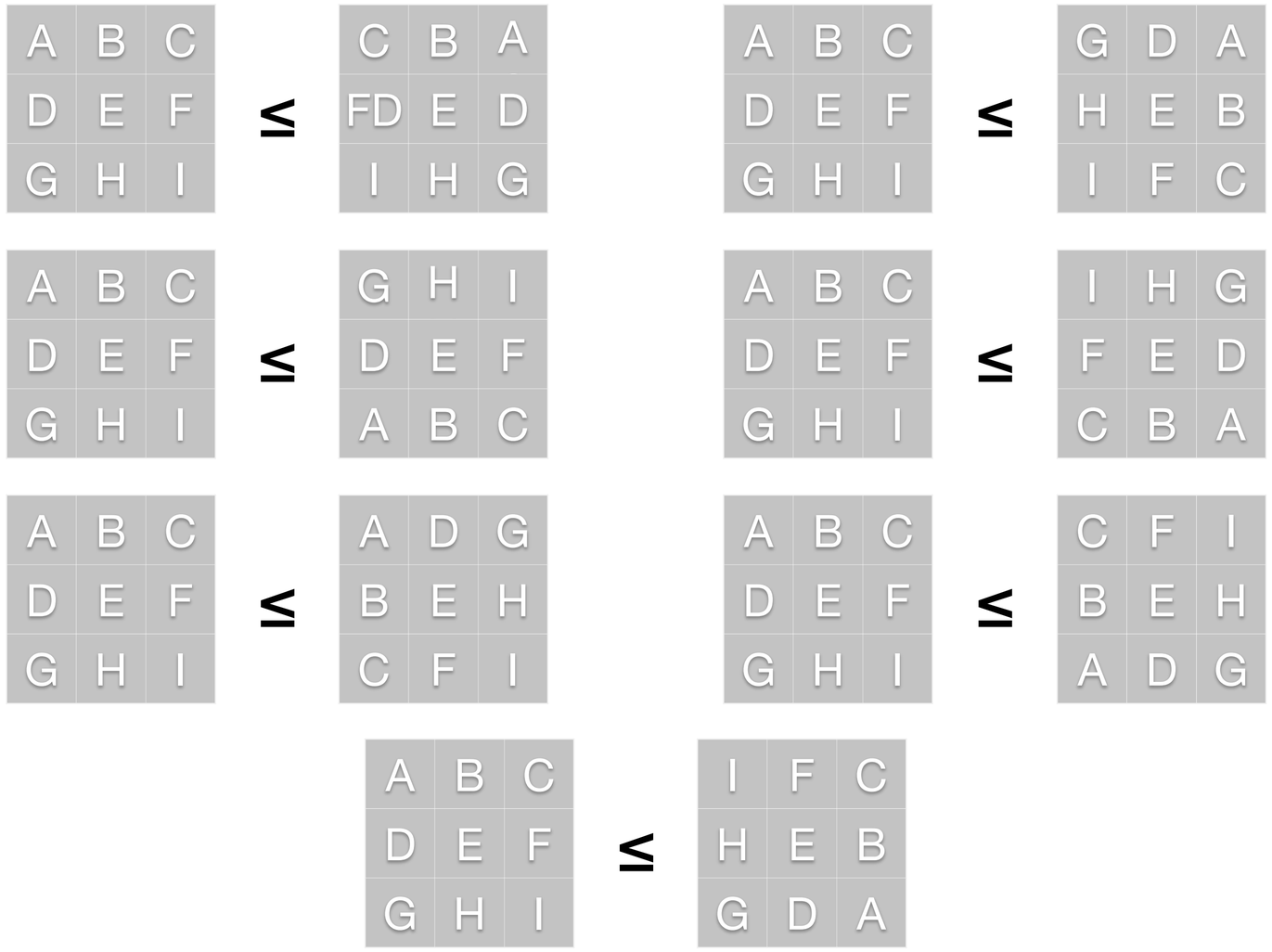
```
A<C \/ (A=C /\
        (B<B \/ (B=B /\
                 (C<A \/ (C=A /\
                          (D<F \/ …)…)
```



1	7	3
9	2	5
6	4	8

which one survives?

Break all symmetries for n=3



Example: non-symmetric Queens

- Find out **how many non-symmetric solutions** exist for n-Queens
- **Trick**: use a Boolean model where `qb[i,j]=true` if and only if there is a queen at position i,j
  (hint: channelling between int and bool model!)
- Add `lex_lesseq` constraints for all symmetric versions of the `qb` board
- Find number of solutions for n=1,2,3,4,5,6,7,8 and check at http://oeis.org

Redundant constraints

- Remember: constraint **modelling** is declarative, constraint **solving is not!**
- Constraints are propagated *one at a time*
- You can sometimes help the solver to propagate better by **adding more constraints**

Redundant constraints

- A constraint is redundant if it is logically *implied* by the model

  - i.e. it does not change the set of solutions

- A constraint may be redundant but **add propagation** (i.e. reduce the search space)

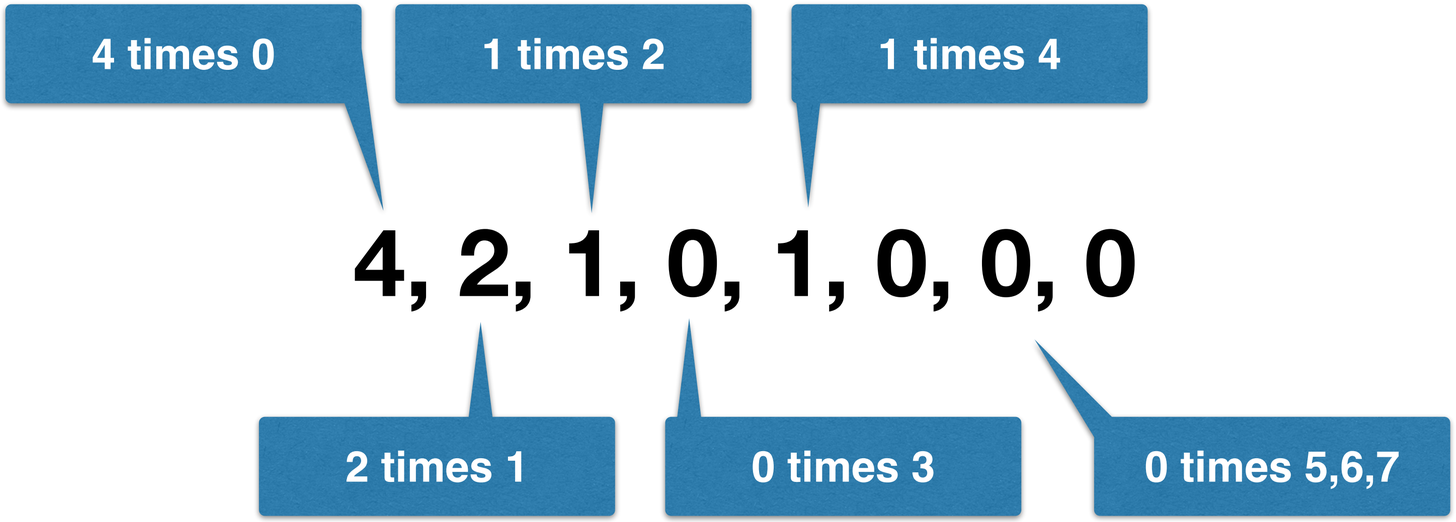
- Cf. global constraints: not redundant but reduce the search space

- Cf. symmetry breaking: not implied, reduce both solution space and search space

Example: magic series

- A magic series is a list of integers $s=[s_0,\ldots,s_{n-1}]$ such that s_i is the number of times i occurs in s

Example: magic series



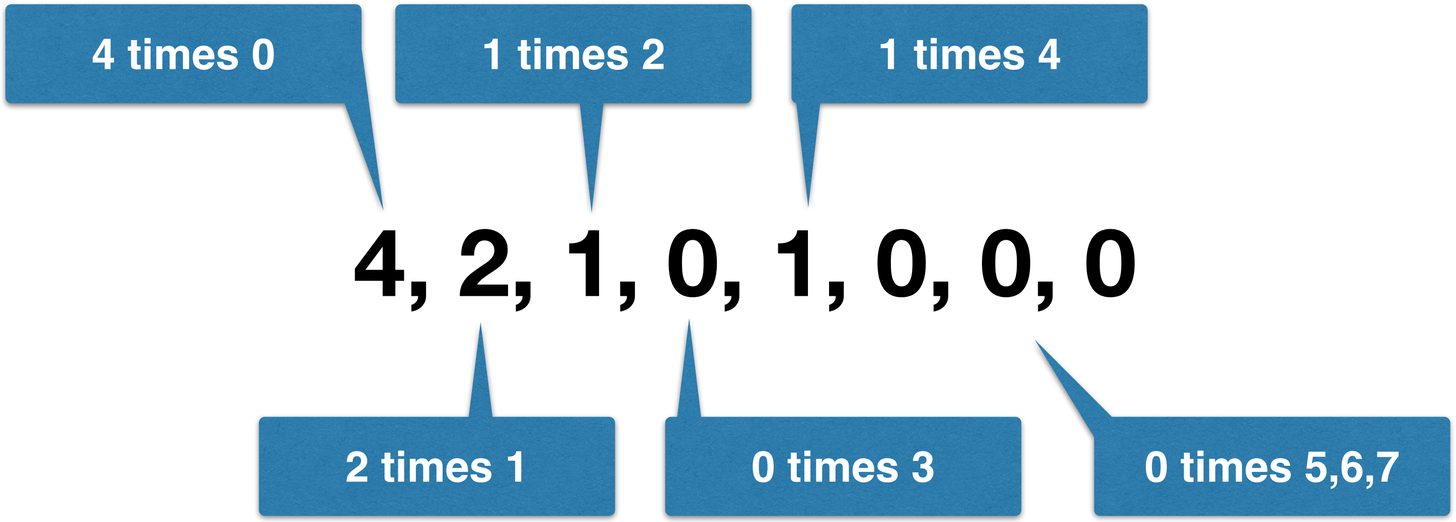
Example: magic series

- A magic series is a list of integers $s=[s_0,\ldots,s_{n-1}]$ such that s_i is the number of time i occurs in s
- Simple model:

```
int: n;
array[0..n-1] of var 0..n: s;
constraint forall (i in 0..n-1) (
  s[i] = sum (j in 0..n-1) (s[j]=i));
```

- Observation 1: how many numbers occur in the series?
  - `sum (s) = n`
- Observation 2: `s[i]*i` is the overall contribution of the i's
  - `sum (i in 0..n-1) (s[i] * i) = n`

Example: magic series



0+0+0+0 + 1+1 + 2 + 4

Example: magic series

- A magic series is a list of integers $s=[s_0,\ldots,s_{n-1}]$ such that s_i is the number of time i occurs in s
- Simple model:

```
int: n;
array[0..n-1] of var 0..n: s;
constraint forall (i in 0..n-1) (
  s[i] = sum (j in 0..n-1) (s[j]=i));
```

- Observation 1: how many numbers occur in the series?
  - `sum (s) = n`
- Observation 2: `s[i]*i` is the overall contribution of the i's
  - `sum (i in 0..n-1) (s[i] * i) = n`


FEDERAL
GALAXY
TOP NEWS
ENLIST
EXIT
MOBILE
INFANTRY
WOULD YOU LIKE TO KNOW MORE?

MiniZinc

- Coursera Courses (Free if you dont want a certificate)
  - Basic Modelling for Discrete Optimization
  - Advanced Modelling for Discrete Optimization
  - Solving Algorithms for Discrete Optimization
  - coursera.org
- MiniZinc Resources
  - Tutorial: https://www.minizinc.org/doc-2.5.5/en/part_2_tutorial.html
  - Reference manual: https://www.minizinc.org/doc-2.5.5/en/part_4_reference.html

MiniZinc

- "Any sufficiently advanced technology is indistinguishable from magic" *Arthur C. Clarke*

- A new way of thinking about problems.

- MiniZinc
  - Thousands of academic and industry users
  - De-facto standard Constraint Programming modelling language