

# Topic 1: Introduction<sup>1</sup>

(Version of 3rd November 2025)

---

Pierre Flener and Jean-Noël Monette

Optimisation Group  
Department of Information Technology  
Uppsala University  
Sweden

Course 1DL451:  
Modelling for Combinatorial Optimisation

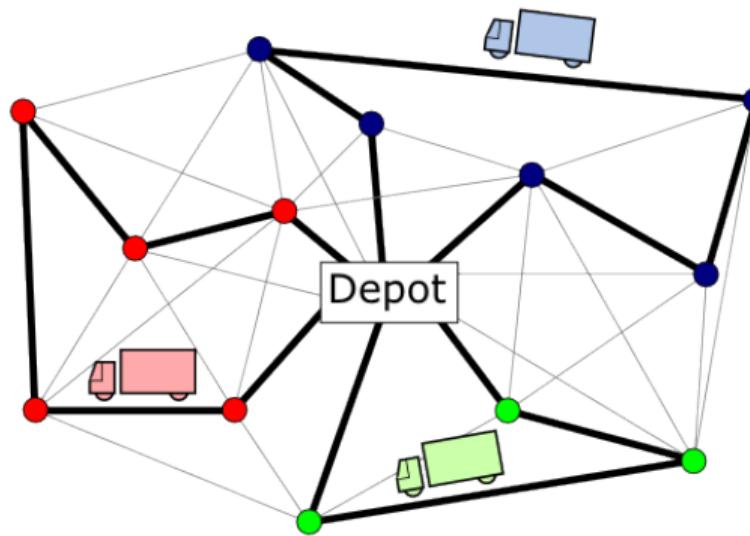
---

<sup>1</sup>Based partly on material by Guido Tack



# Optimisation

- Constraint Problems
- Combinatorial Optimisation
- Modelling (in MiniZinc)
- Solving
- The MiniZinc Toolchain
- Course Information



Optimisation is a science of **service**:  
to scientists, to engineers, to artists, and to society.



# Slogan of the Course

---

Constraint  
Problems

Combinatorial  
Optimisation

Modelling  
(in MiniZinc)

Solving

The MiniZinc  
Toolchain

Course  
Information

How to solve problems  
without knowing how to solve problems?



## MiniZinc Challenge 2024: 16 (of 20) Problems and Winners

Problem and Model	Backend and Solver	Solving Technology
aircraft-disassembly	CP-SAT (by Google)	portfolio: LCG, MIP, CBLS
cable-tree-wiring	CP-SAT (by Google)	portfolio: LCG, MIP, CBLS
community-detection	MZN/Gurobi	MIP
compression	PicatSAT	SAT
concert-hall-cap	MZN/Gurobi	MIP
fox-geese-corn	MZN/Gurobi	MIP
hoist-benchmark	Chuffed	hybrid: LCG = CP + SAT
monitor-placement-1id	CP Optimizer (by IBM)	CP
neighbours	CP-SAT (by Google)	portfolio: LCG, MIP, CBLS
peaceable_queens	Gecode-Dexter	portfolio: CP, LNS
portal	Chuffed	hybrid: LCG = CP + SAT
tiny-cvrp	MZN/Gurobi	MIP
train-scheduling	CP-SAT (by Google)	portfolio: LCG, MIP, CBLS
word-equations	PicatSAT	SAT
yumi-dynamic	CP-SAT (by Google)	portfolio: LCG, MIP, CBLS



# Outline

---

**1. Constraint Problems**

**2. Combinatorial Optimisation**

**3. Modelling (in MiniZinc)**

**4. Solving**

**5. The MiniZinc Toolchain**

**6. Course Information**



# Outline

---

## 1. Constraint Problems

## 2. Combinatorial Optimisation

## 3. Modelling (in MiniZinc)

## 4. Solving

## 5. The MiniZinc Toolchain

## 6. Course Information



## Example (Agricultural experiment design)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley							
corn							
millet							
oats							
rye							
spelt							
wheat							

### Constraints to be satisfied:

- 1 Equal growth load: Every plot grows 3 grains.
- 2 Equal sample size: Every grain is grown in 3 plots.
- 3 Balance: Every grain pair is grown in 1 common plot.

**Instance:** 7 plots, 7 grains, 3 grains/plot, 3 plots/grain, balance 1.



## Example (Agricultural experiment design)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	✓	✓	✓	—	—	—	—
corn	✓	—	—	✓	✓	—	—
millet	✓	—	—	—	—	✓	✓
oats	—	✓	—	✓	—	✓	—
rye	—	✓	—	—	✓	—	✓
spelt	—	—	✓	✓	—	—	✓
wheat	—	—	✓	—	✓	✓	—

### Constraints to be satisfied:

- 1 Equal growth load: Every plot grows 3 grains.
- 2 Equal sample size: Every grain is grown in 3 plots.
- 3 Balance: Every grain pair is grown in 1 common plot.

**Instance:** 7 plots, 7 grains, 3 grains/plot, 3 plots/grain, balance 1.



## Example (Doctor rostering)

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Doctor A							
Doctor B							
Doctor C							
Doctor D							
Doctor E							

### Constraints to be satisfied:

- 1 #on-call doctors / day = 1
- 2 #operating doctors / weekday  $\leq 2$
- 3 #operating doctors / week  $\geq 7$
- 4 #appointed doctors / week  $\geq 4$
- 5 day off after operation day
- 6 ...

Objective function to be minimised: Cost: ...



## Example (Doctor rostering)

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Doctor A	call	none	oper	none	oper	none	none
Doctor B	appt	call	none	oper	none	none	call
Doctor C	oper	none	call	appt	appt	call	none
Doctor D	appt	oper	none	call	oper	none	none
Doctor E	oper	none	oper	none	call	none	none

**Constraints to be satisfied:**

- 1 #on-call doctors / day = 1
- 2 #operating doctors / weekday  $\leq 2$
- 3 #operating doctors / week  $\geq 7$
- 4 #appointed doctors / week  $\geq 4$
- 5 day off after operation day
- 6 ...



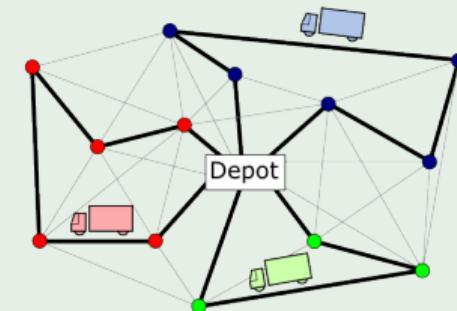
**Objective function to be minimised:** Cost: ...

## Example (Vehicle routing: parcel delivery)

**Given** a depot with parcels for clients and a vehicle fleet,  
**find** which vehicle visits which client when.

**Constraints** to be **satisfied**:

- 1 All parcels are delivered on time.
- 2 No vehicle is overloaded.
- 3 Driver regulations are respected.
- 4 ...



**Objective function** to be **minimised**:

- Cost: the total fuel consumption and driver salary.

## Example (Travelling salesperson: optimisation TSP)

**Given** a map and cities,  
**find** a **shortest** route visiting each city once and returning to the starting city.



# Applications in Air Traffic Management

Constraint  
Problems

Combinatorial  
Optimisation

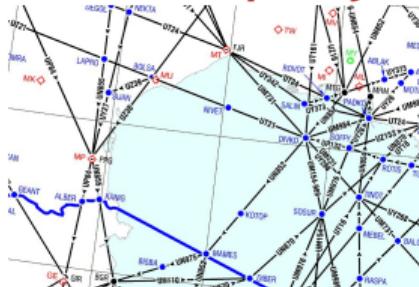
Modelling  
(in MiniZinc)

Solving

The MiniZinc  
Toolchain

Course  
Information

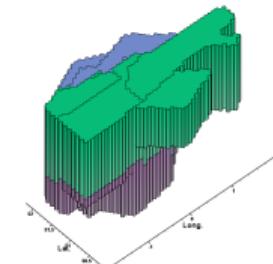
## Demand vs capacity



## Contingency planning

Flow	Time Span	Hourly Rate
From: Arlanda	00:00 – 09:00	3
	09:00 – 18:00	5
	18:00 – 24:00	2
From: Arlanda	00:00 – 12:00	4
	12:00 – 24:00	3
...	...	...

## Airspace sectorisation

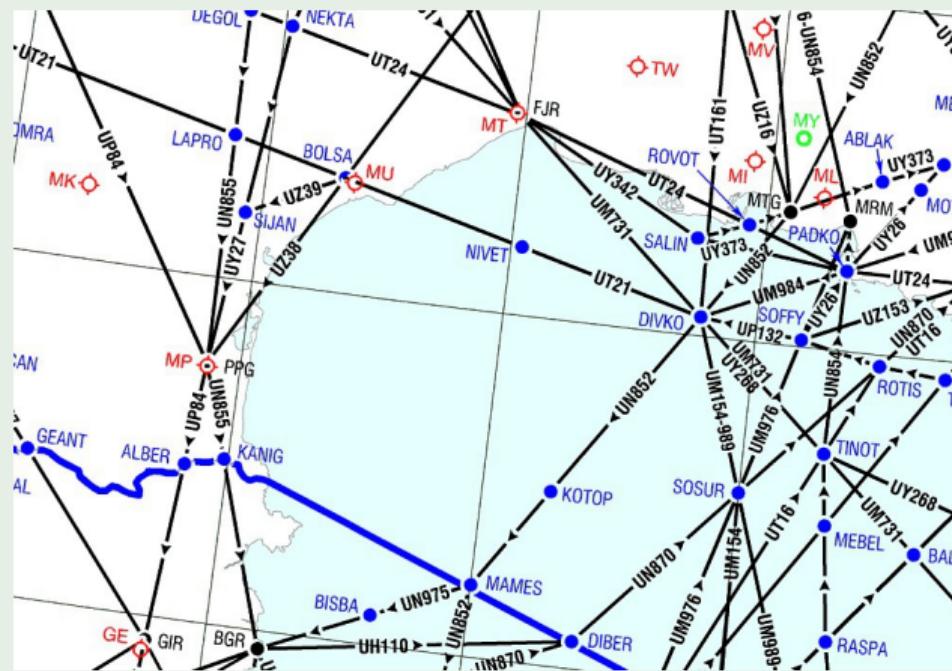


## Workload balancing



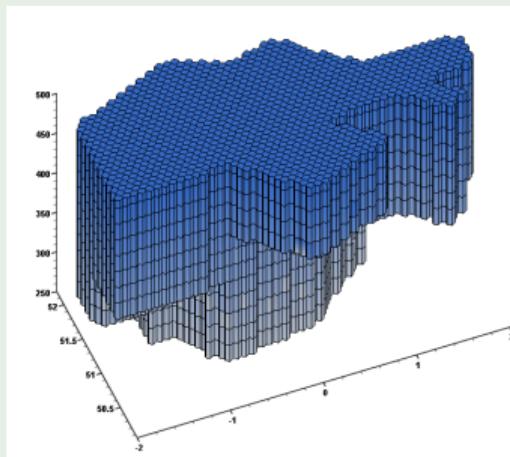
### Example (Air-traffic demand-capacity balancing)

Reroute flights, in height and speed, so as to balance the workload of air traffic controllers in a multi-sector airspace:

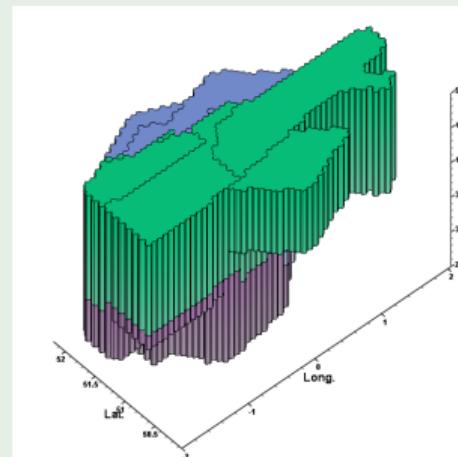


## Example (Airspace sectorisation)

**Given** an airspace split into  $c$  cells, a targeted number  $s$  of sectors, and flight schedules.



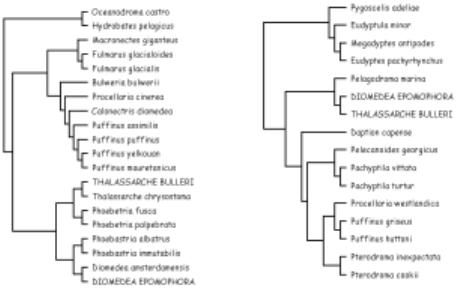
**Find** a colouring of the  $c$  cells into  $s$  connected convex sectors, with minimal imbalance of the workloads of their air traffic controllers.



There are  $s^c$  possible colourings, but very few optimally satisfy the constraints: is **intelligent** search necessary?

# **Applications in Biology and Medicine**

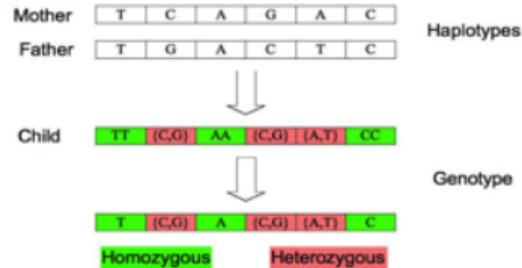
## Phylogenetic supertree



# Medical image analysis



## Haplotype inference

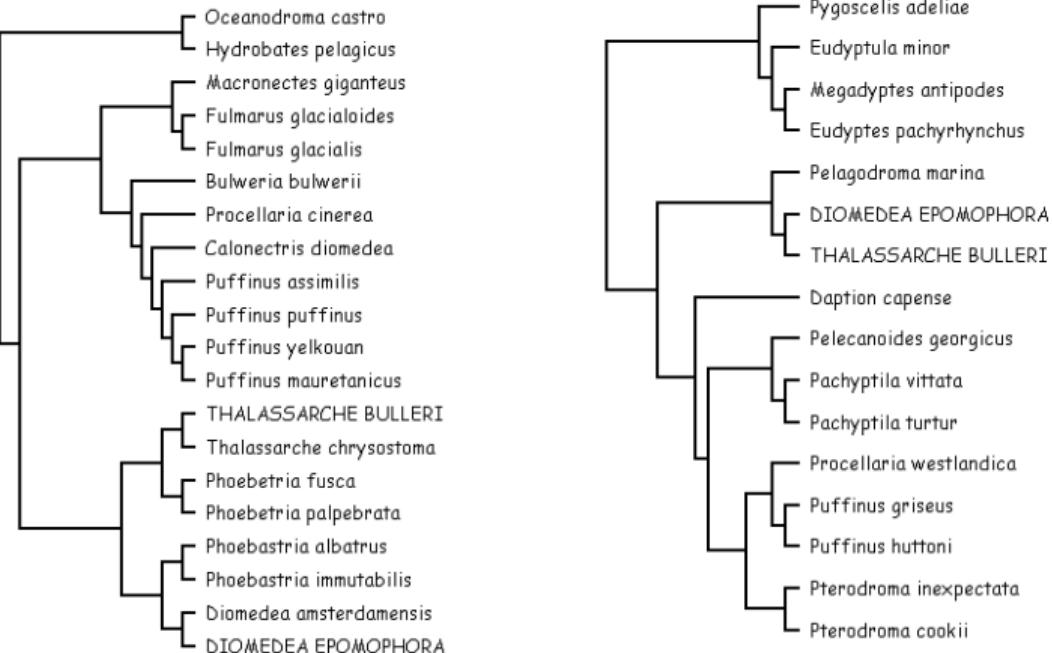


## Doctor rostering





# Example (What supertree is maximally consistent with several given trees that share some species?)





## Example (Haplotype inference by pure parsimony)

**Given**  $n$  child genotypes, with homo- and heterozygous sites:

A	C / G	T	C	A / T	C
...					
A / T	G	T	C / G	A	C
...					

**find** a minimal set of (at most  $2 \cdot n$ ) parent haplotypes:

A	C	T	C	T	C
...					
A	G	T	C	A	C
...					
T	G	T	G	A	C
...					

**so that** each given genotype conflates 2 found haplotypes.



# Applications in Programming and Testing

Constraint  
Problems

Combinatorial  
Optimisation

Modelling  
(in MiniZinc)

Solving

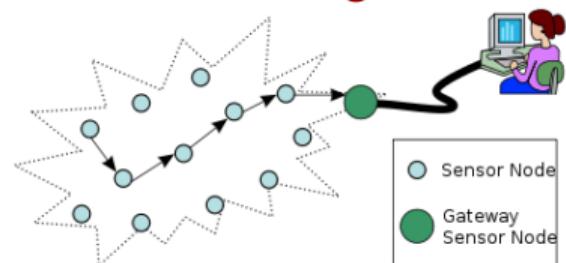
The MiniZinc  
Toolchain

Course  
Information

## Robot programming



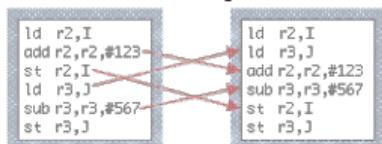
## Sensor-net configuration



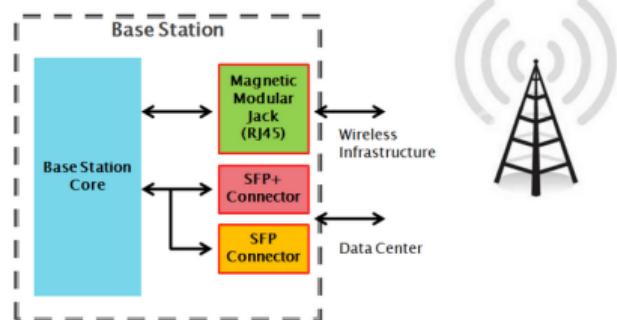
## Compiler design

COMPILERS  
FOR INSTRUCTION SCHEDULING

C Compiler  
C++ Compiler



## Base-station testing





# Other Application Areas

Constraint  
Problems

Combinatorial  
Optimisation

Modelling  
(in MiniZinc)

Solving

The MiniZinc  
Toolchain

Course  
Information

## School timetabling

	Monday	Tuesday	Wednesday	Thursday	Friday
9:00	KMF200 Ordinary Differential Equations I FTII		LNUC2007 Computer Graphics (I) Dual	KMF2003 Numerical Analysis I Williamsson, GÖS	
10:00				KMF2002 Ordinary Differential Equations Simon Engineering, Basement Theatre M. Williamsson, GÖS	KMF2002 Ordinary Differential Equations HÖST
11:00	CSD1010 Algorithm and Data Structures 1.1		KMF2012 Further Linear Algebra 1.1		KMF2012 Ordinary Differential Equations Williamsson, GÖS
12:00	HMF2112 Further Linear Algebra Reson, Theatre A	KMF2002 Numerical Analysis I Williamsson, GÖS	CSD1012 Computer Graphics 1.1		KMF2012 Further Linear Algebra Stephant, Theatre 1
13:00			PASS Peer-Assisted Study MET-LJTFB-LJTFZ-HÖST		KMF2012 Further Linear Algebra Stephant, Theatre 1
14:00	CSD1012 Computer Graphics 1.1			KMF2012 Further Linear Algebra HÖST	
15:00		CSTUF Tutorial			
16:00	CSD1012 Algorithm and Data Structures 1.1				

## Security: SQL injection?



## Sports tournament design

svensk handball



## Container packing





## Definitions

In a **constraint problem**, values have to be **found** for all the unknowns, called **variables** (in the mathematical sense; also called **decision variables**) and ranging over **given** sets, called **domains**, so that:

- All the given **constraints** on the decision variables are **satisfied**.
- Optionally: A given **objective function** on the decision variables has an optimal value: either a **minimal** cost or a **maximal** profit.

A **candidate solution** to a constraint problem maps each decision variable to a value within its domain; it is:

- **feasible** if all the constraints are satisfied;
- **optimal** if the objective function takes an optimal value.

The **search space** consists of all candidate solutions.

A **solution** to a **satisfaction problem** is feasible.

An **optimal solution** to an **optimisation problem** is feasible and optimal.



This is one of the seven **Millennium Prize** problems of the Clay Mathematics Institute (Massachusetts, USA), each worth 1 million US\$.

Informally:

- P = class of problems that need **no** search to be solved  
NP = class of problems that **might** need search to solve
- P = class of problems with **easy-to-compute** solutions  
NP = class of problems with **easy-to-check** solutions

Thus: Can search always be avoided ( $P = NP$ ),  
or is search sometimes necessary ( $P \neq NP$ )?

Problems that are solvable in polynomial time (in the input size) are considered **tractable**, aka **easy**.

Problems needing super-polynomial time are considered **intractable**, aka **hard**.



# NP Completeness: Examples

Given a digraph  $(V, E)$ :

## Examples

- Finding a **shortest path** takes  $\mathcal{O}(V \cdot E)$  time and is thus in P.
- Determining the existence of a simple path (which has distinct vertices), from a given single source, that has *at least* a given number  $\ell$  of edges is NP-complete. Hence finding a **longest path** seems hard:  
increase  $\ell$  starting from a trivial lower bound, until answer is ‘no’.

## Examples

- Finding an **Euler tour** (which visits each *edge* once) takes  $\mathcal{O}(E)$  time and is thus in P.
- Determining the existence of a **Hamiltonian cycle** (which visits each *vertex* once) is NP-complete.



# NP Completeness: More Examples

## Examples

- ***n-SAT***: Determining the satisfiability of a conjunction of disjunctions of  $n$  Boolean literals is in P for  $n = 2$  but NP-complete for  $n = 3$ .
- ***SAT***: Determining the satisfiability of a formula over Boolean literals is NP-complete.
- ***Clique***: Determining the existence of a clique (complete subgraph) of a given size in a graph is NP-complete.
- ***Vertex Cover***: Determining the existence of a vertex cover (a vertex subset with at least one endpoint for all edges) of a given size in a graph is NP-complete.
- ***Subset Sum***: Determining the existence of a subset, of a given set, that has a given sum is NP-complete.



## Search spaces are often larger than the universe!



Many important real-life problems are NP-hard or worse: their real-life instances can only be solved exactly and fast enough by **intelligent** search, unless  $P = NP$ . **NP-hardness is not where the fun ends, but where it begins!**



## Example (Optimisation TSP over $n$ cities)

A brute-force algorithm evaluates all  $n!$  candidate routes:

- A computer of today evaluates  $10^6$  routes / second:

$n$	time
11	40 seconds
14	1 day
18	203 years
20	77k years

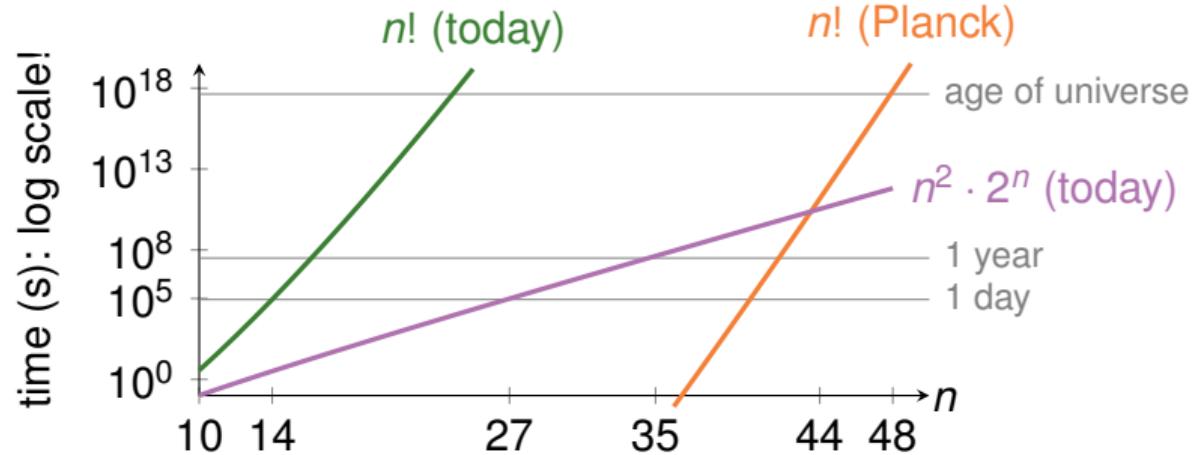
- Planck time is shortest useful interval:  $\approx 5.4 \cdot 10^{-44}$  second; a Planck computer would evaluate  $1.8 \cdot 10^{43}$  routes / second:

$n$	time
37	0.7 seconds
41	20 days
48	1.5 · age of universe

The dynamic program by Bellman-Held-Karp “only” takes  $\mathcal{O}(n^2 \cdot 2^n)$  time: a computer of today takes a day for  $n = 27$ , a year for  $n = 35$ , the age of the universe for  $n = 67$ , and beats the  $\mathcal{O}(n!)$  algo on Planck computer for  $n \geq 44$ .

# Intelligent Search upon NP-Hardness

Do not give up but try to stay ahead of the curve:  
there is an instance size until which an **exact** algorithm is fast enough!



Concorde TSP Solver beats the Bellman-Held-Karp exact algo: it uses local search & approximation algos, but sometimes proves exactness of its optima.  
The largest instance solved exactly, in 136 CPU years in 2006, has  $n = 85900$ .



# Outline

---

1. Constraint Problems

2. Combinatorial Optimisation

3. Modelling (in MiniZinc)

4. Solving

5. The MiniZinc Toolchain

6. Course Information



A **solving technology** offers languages, methods, and tools for:

**what:** **Modelling** constraint problems in a **declarative** language.

and / or

**how:** **Solving** constraint problems **intelligently**:

- **Search**: Explore the space of candidate solutions.
- **Reasoning**: Reduce the space of candidate solutions.
- **Relaxation**: Exploit solutions to easier problems.

A **solver** is a program that takes a model and data as input and tries to solve that problem instance.

Combinatorial (= discrete) optimisation covers satisfaction *and* optimisation problems for variables ranging over *discrete* sets: **combinatorial problems**.

The ideas in this course extend to continuous optimisation, to soft optimisation, and to stochastic optimisation.



## Examples (Solving technologies)

With general-purpose solvers, taking model and data as input:

- Boolean satisfiability (SAT)
- SAT (resp. optimisation) modulo theories (SMT and OMT)
- (Mixed) integer linear programming (IP and MIP)
- Constraint programming (CP) ☞ available via [1DL705](#)
- ...
- Hybrid technologies ( $LCG = CP + SAT, \dots$ ) and portfolios

Methodologies, *usually without* modelling and solvers:

- Dynamic programming (DP)
- Greedy algorithms
- Approximation algorithms
- Local search (LS)
- ...



# Outline

---

1. Constraint Problems

2. Combinatorial Optimisation

3. Modelling (in MiniZinc)

4. Solving

5. The MiniZinc Toolchain

6. Course Information



# What vs How

## Example

Consider the **problem** of sorting an array  $A$  of  $n$  numbers into an array  $S$  of increasing-or-equal numbers.

A **formal specification** is:

$$\text{sort}(A, S) \equiv \text{permutation}(A, S) \wedge \text{increasing}(S)$$

saying that  $S$  must be a permutation of  $A$  in increasing order.

Seen as a generate-and-test **algorithm**, it takes  $\mathcal{O}(n!)$  time, but it can be refined into the existing  $\mathcal{O}(n \log n)$  algorithms.

A **specification** is a **declarative** description of **what** problem is to be solved. An **algorithm** is an **imperative** description of **how** to solve the problem (fast).



# Modelling vs Programming

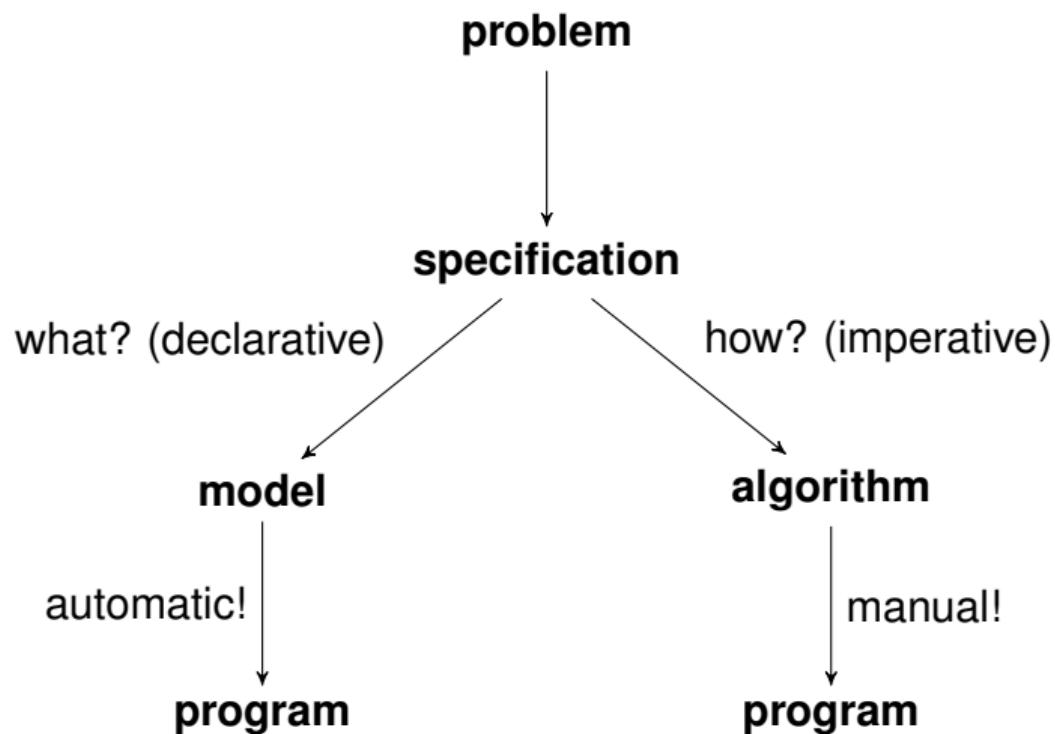
Constraint  
Problems

Combinatorial  
Optimisation

Modelling  
(in MiniZinc)

Solving  
The MiniZinc  
Toolchain

Course  
Information





# Example (Sudoku)

Constraint  
ProblemsCombinatorial  
OptimisationModelling  
(in MiniZinc)

Solving

The MiniZinc  
ToolchainCourse  
Information

8								
	3	6						
7		9		2				
5			7					
		4	5	7				
		1			3			
	1				6	8		
	8	5			1			
9				4				

8	1	2	7	5	3	6	4	9
9	4	3	6	8	2	1	7	5
6	7	5	4	9	1	2	8	3
1	5	4	2	3	7	8	9	6
3	6	9	8	4	5	7	2	1
2	8	7	1	6	9	5	3	4
5	2	1	9	7	4	3	6	8
4	3	8	5	2	6	9	1	7
7	9	6	3	1	8	4	5	2

A Sudoku is a 9-by-9 array of integers in the range 1..9. Some of the elements are provided as parameters. The remaining elements are unknowns that have to satisfy the following constraints:

- 1 the elements in each row are all different;
- 2 the elements in each column are all different;
- 3 the elements in each 3-by-3 block are all different.



## Example (Sudoku)

Google

Translate

Turn off instant

English Turkish Swedish English - detected ▾



MiniZinc Turkish Swedish ▾

Translate

A Sudoku is a 9-by-9 array of integers in the interval 1..9. ×  
Some of the elements are provided as parameters.

The remaining elements are unknowns  
that have to satisfy the following constraints:  
- the elements in each row are all different;  
- the elements in each column are all different;  
- the elements in each 3-by-3 block are all different.

```
array[1..9,1..9] of var 1..9: Sudoku;
solve satisfy;
constraint forall(row in 1..9)
  (alldifferent(Sudoku[row, ..]));
constraint forall(col in 1..9)
  (alldifferent(Sudoku[.., col]));
constraint forall(i,j in {0,3,6})
  (alldifferent(Sudoku[i+1..i+3, j+1..j+3]));
```



## Example (Sudoku ↗)

8								
	3	6						
7			9		2			
5				7				
			4	5	7			
		1				3		
	1					6	8	
	8	5				1		
9				4				

8	1	2	7	5	3	6	4	9
9	4	3	6	8	2	1	7	5
6	7	5	4	9	1	2	8	3
1	5	4	2	3	7	8	9	6
3	6	9	8	4	5	7	2	1
2	8	7	1	6	9	5	3	4
5	2	1	9	7	4	3	6	8
4	3	8	5	2	6	9	1	7
7	9	6	3	1	8	4	5	2

```
-2 array[1..9,1..9] of var 1..9: Sudoku;
-1 ... % load the hints
0 solve satisfy;
1 constraint forall(row in 1..9) (all_different(Sudoku[row,...]));
2 constraint forall(col in 1..9) (all_different(Sudoku[...,col]));
3 constraint forall(i,j in {0,3,6})
    (all_different(Sudoku[i+1..i+3,j+1..j+3]));
```



## Example (Agricultural experiment design, AED)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	✓	✓	✓	—	—	—	—
corn	✓	—	—	✓	✓	—	—
millet	✓	—	—	—	—	✓	✓
oats	—	✓	—	✓	—	✓	—
rye	—	✓	—	—	✓	—	✓
spelt	—	—	✓	✓	—	—	✓
wheat	—	—	✓	—	✓	✓	—

**Constraints to be satisfied:**

- 1 Equal growth load: Every plot grows 3 grains.
- 2 Equal sample size: Every grain is grown in 3 plots.
- 3 Balance: Every grain pair is grown in 1 common plot.

**Instance:** 7 plots, 7 grains, 3 grains/plot, 3 plots/grain, balance 1.

General term: balanced incomplete block design (BIBD).



## Example (Agricultural experiment design, AED)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	1	1	1	0	0	0	0
corn	1	0	0	1	1	0	0
millet	1	0	0	0	0	1	1
oats	0	1	0	1	0	1	0
rye	0	1	0	0	1	0	1
spelt	0	0	1	1	0	0	1
wheat	0	0	1	0	1	1	0

**Constraints to be satisfied:**

- 1 Equal growth load: Every plot grows 3 grains.
- 2 Equal sample size: Every grain is grown in 3 plots.
- 3 Balance: Every grain pair is grown in 1 common plot.

**Instance:** 7 plots, 7 grains, 3 grains/plot, 3 plots/grain, balance 1.

General term: balanced incomplete block design (BIBD).



In a BIBD, the plots are called **blocks** and the grains are called **varieties**:

### Example (BIBD *integer* model ↗: ✓ ↵ 1 and – ↵ 0)

```
-3 enum Varieties; enum Blocks;
-2 int: blockSize; int: sampleSize; int: balance;
-1 array[Varieties,Blocks] of var 0..1: BIBD; % BIBD[v,b]=1 iff v is in b
0 solve satisfy;
1 constraint forall(b in Blocks) (blockSize = sum(BIBD[.,b]));
2 constraint forall(v in Varieties) (sampleSize = sum(BIBD[v,.]));
3 constraint forall(v, w in Varieties where v < w)
    (balance = sum([BIBD[v,b]*BIBD[w,b] | b in Blocks]));
```

### Example (Instance data for our AED ↗)

```
-3 Varieties = {barley,...,wheat}; Blocks = {plot1,...,plot7};
-2 blockSize = 3; sampleSize = 3; balance = 1;
```

Using the `count` abstraction instead of `sum`:Example (BIBD *integer* model ↗: ✓ ↵ 1 and – ↵ 0)

```
-3 enum Varieties; enum Blocks;
-2 int: blockSize; int: sampleSize; int: balance;
-1 array[Varieties,Blocks] of var 0..1: BIBD; % BIBD[v,b]=1 iff v is in b
0 solve satisfy;
1 constraint forall(b in Blocks) (blockSize = count(BIBD[.,b], 1));
2 constraint forall(v in Varieties) (sampleSize = count(BIBD[v,.], 1));
3 constraint forall(v, w in Varieties where v < w)
    (balance = count([BIBD[v,b]*BIBD[w,b] | b in Blocks], 1));
```

## Example (Instance data for our AED ↗)

```
-3 Varieties = {barley,...,wheat}; Blocks = {plot1,...,plot7};
-2 blockSize = 3; sampleSize = 3; balance = 1;
```

Using the `count` abstraction over linear expressions:Example (BIBD *integer* model ↗: ✓ ↵ 1 and – ↵ 0)

```
-3 enum Varieties; enum Blocks;
-2 int: blockSize; int: sampleSize; int: balance;
-1 array[Varieties,Blocks] of var 0..1: BIBD; % BIBD[v,b]=1 iff v is in b
0 solve satisfy;
1 constraint forall(b in Blocks) (blockSize = count(BIBD[.,b], 1));
2 constraint forall(v in Varieties) (sampleSize = count(BIBD[v,.], 1));
3 constraint forall(v, w in Varieties where v < w)
    (balance = count([BIBD[v,b]+BIBD[w,b] | b in Blocks], 2));
```

## Example (Instance data for our AED ↗)

```
-3 Varieties = {barley,...,wheat}; Blocks = {plot1,...,plot7};
-2 blockSize = 3; sampleSize = 3; balance = 1;
```



## Reconsider the model fragment:

```
2 constraint forall(v in Varieties) (sampleSize = count(BIBD[v,..], 1));
```

This constraint is **declarative** (and by the way non-linear),  
so read it using only the verb “to be” or synonyms thereof:

*for all varieties v,  
the count of occurrences of 1 in row v of BIBD  
must equal sampleSize*

The constraint is **not procedural**:

*for all varieties v,  
we first count the occurrences of 1 in row v  
and then check if that count equals sampleSize*

The latter reading is appropriate for solution **checking**,  
but solution **finding** performs no such procedural counting.



## Example (Idea for another BIBD model)

barley	{plot1, plot2, plot3}	}
corn	{plot1, plot4, plot5}	}
millet	{plot1, plot6, plot7}	}
oats	{plot2, plot4, plot6}	}
rye	{plot2, plot5, plot7}	}
spelt	{plot3, plot4, plot7}	}
wheat	{plot3, plot5, plot6}	}

### Constraints to be satisfied:

- 1 Equal growth load: Every plot grows 3 grains.
- 2 Equal sample size: Every grain is grown in 3 plots.
- 3 Balance: Every grain pair is grown in 1 common plot.



## Example (BIBD set model ↗: a block set per variety)

```
-3 enum Varieties; enum Blocks;  
-2 int: blockSize; int: sampleSize; int: balance;  
-1 array[Varieties] of var set of Blocks: BIBD; % BIBD[v] = blocks for v  
0 solve satisfy;  
1 constraint forall(b in Blocks)  
    (blockSize = sum(v in Varieties) (b in BIBD[v]));  
2 constraint forall(v in Varieties)  
    (sampleSize = card(BIBD[v]));  
3 constraint forall(v, w in Varieties where v < w)  
    (balance = card(BIBD[v] intersect BIBD[w]));
```

## Example (Instance data for our AED ↗)

```
-3 Varieties = {barley, ..., wheat}; Blocks = {plot1, ..., plot7};  
-2 blockSize = 3; sampleSize = 3; balance = 1;
```



## Example (Doctor rostering)

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Doctor A	call	none	oper	none	oper	none	none
Doctor B	appt	call	none	oper	none	none	call
Doctor C	oper	none	call	appt	appt	call	none
Doctor D	appt	oper	none	call	oper	none	none
Doctor E	oper	none	oper	none	call	none	none

**Constraints to be satisfied:**

- 1 #on-call doctors / day = 1
- 2 #operating doctors / weekday  $\leq 2$
- 3 #operating doctors / week  $\geq 7$
- 4 #appointed doctors / week  $\geq 4$
- 5 day off after operation day
- 6 ...



**Objective function to be minimised:** Cost: ...



## Example (Doctor rostering ↗)

```
-5 set of int: Days; % d mod 7 = 1 iff d is a Monday
-4 enum Doctors;
-3 enum ShiftTypes = {appt, call, oper, none};
-2 % Roster[i,j] = shift type of Dr i on day j:
-1 array[Doctors,Days] of var ShiftTypes: Roster;
0 solve minimize ...; % plug in an objective function
1 constraint forall(d in Days) (count(Roster[..,d],call) = 1);
2 constraint forall(d in Days where d mod 7 in 1..5)
    (count(Roster[..,d],oper) <= 2);
3 constraint count(Roster,oper) >= 7;
4 constraint count(Roster,appt) >= 4;
5 constraint forall(d in Doctors)
    (regular(Roster[d,..],"((oper none) | appt | call | none)*"));
6 ... % other constraints
```

## Example (Instance data for our small hospital unit ↗)

```
-5 Days = 1..7;
-4 Doctors = {Dr_A, Dr_B, Dr_C, Dr_D, Dr_E};
```



## Using decision variables as indices within arrays: **black magic?**

### Example (Job allocation at minimal salary cost)

**Given** jobs `Jobs` and the salaries of work applicants `Apps`,  
**find** a work applicant for each job

**such that** some constraints (on the qualifications of the work applicants for the jobs, on workload distribution, etc) are satisfied and the total salary cost is minimal:

```
1 array[Apps] of 0..1000: Salary; % Salary[a] = cost per job to appl. a
2 array[Jobs] of var Apps: Worker; % Worker[j] = appl. allocated job j
3 solve minimize sum(j in Jobs)(Salary[Worker[j]]); 
4 constraint ...; % qualifications, workload, etc
```



# Using decision variables as indices within arrays: black magic?!

## Example (Vehicle routing: backbone model)

```
enum Cities = {AMS, BRU, LUX, CDG}
```

Next: 

AMS	BRU	LUX	CDG





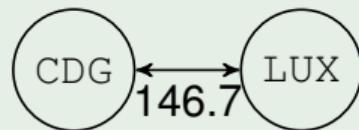
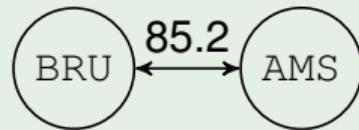
# Using decision variables as indices within arrays: black magic?!

## Example (Vehicle routing: backbone model)

```
enum Cities = {AMS, BRU, LUX, CDG}
```

	AMS	BRU	LUX	CDG
Next:	BRU	AMS	CDG	LUX

So `all_different(Next)` is too weak!





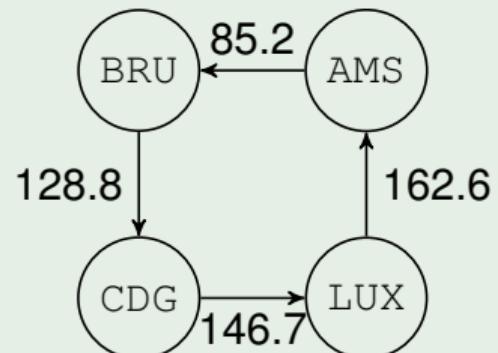
# Using decision variables as indices within arrays: black magic?!

## Example (Vehicle routing: backbone model)

```
enum Cities = {AMS, BRU, LUX, CDG}
```

	AMS	BRU	LUX	CDG
Next:	BRU	CDG	AMS	LUX

Let us use `circuit` (`Next`) instead:





# Using decision variables as indices within arrays: black magic?

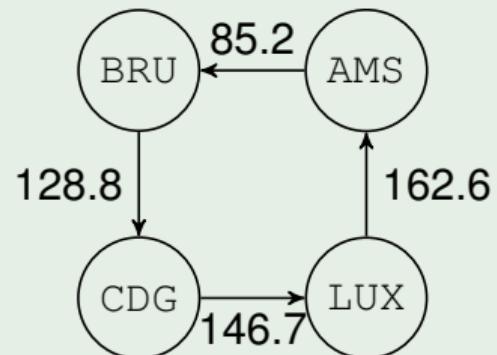
## Example (Vehicle routing: backbone model)

```
enum Cities = {AMS, BRU, LUX, CDG}
```

	AMS	BRU	LUX	CDG
Next:	BRU	CDG	AMS	LUX

Let us use `circuit` (`Next`) instead:

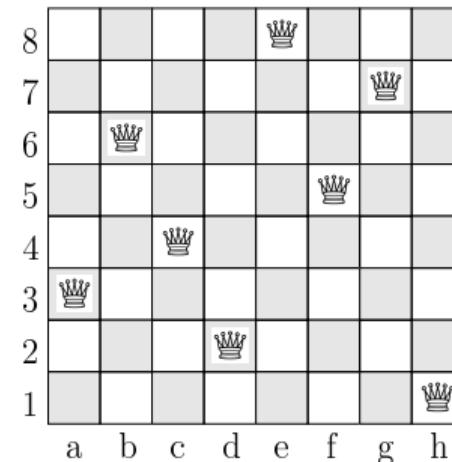
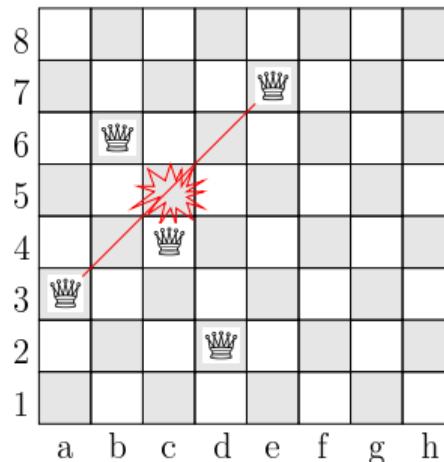
```
1 array[Cities,Cities] of float: Distance; % instance data
2 array[Cities] of var Cities: Next; % travel from c to Next[c]
3 solve minimize sum(c in Cities) (Distance[c,Next[c]]);
4 constraint circuit(Next);
5 constraint ...; % side constraints, if any
```





# Toy Example: 8-Queens

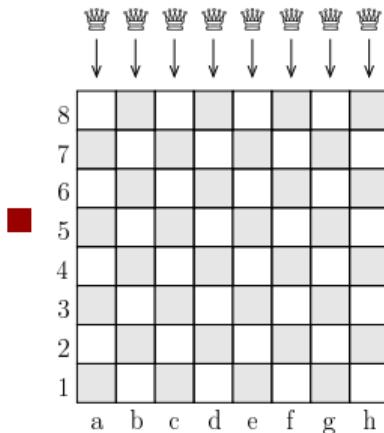
Can one place 8 queens onto an  $8 \times 8$  chessboard so that all queens are in distinct rows, columns, and diagonals?





# An 8-Queens Model

One of the many models, with **one** decision variable per queen:



Let **decision variable**  $\text{Row}[c]$ , of **domain**  $1 \dots 8$ , denote the row of the queen in column  $c$ , for  $c$  in  $\{a, b, c, \dots, h\}$ , which we rename into  $1 \dots 8$ . Example:  $\text{Row}[3] = 4$  means that the queen of column 3 (column  $c$  in the picture) is in row 4. The **constraint** that all queens must be in distinct columns is **satisfied** by the choice of variables!

■ The remaining **constraints** to be **satisfied** are:

- All queens are in distinct rows: the var.s  $\text{Row}[c]$  take distinct values for all  $c$
- All queens are in distinct diagonals:  
the expressions  $\text{Row}[c] + c$  take distinct values for all  $c$   
the expressions  $\text{Row}[c] - c$  take distinct values for all  $c$



# An 8-Queens Model in MiniZinc

Consider the following model [↗](#) in a file 8-queens.mzn:

```
1 include "globals.mzn"; % ensures that lines 4 to 6 are understood
2 int: n = 8; % the given number of queens
3 array[1..n] of var 1..n: Row; % Row[c] = the unknown row of the queen
      in column c;      enforces that all queens are in distinct columns
4 constraint all_different( Row ) ; % distinct rows
5 constraint all_different([Row[c]+c | c in 1..n]); % distinct up-dia.
6 constraint all_different([Row[c]-c | c in 1..n]); % distinct down-dia.
7 solve satisfy; % solve to satisfaction of all the constraints
8 output [show(Row)]; % pretty-printing of solutions
```

The `all_different(X)` constraint holds if and only if all the expressions in the array `X` take different values.



# Modelling Concepts

- A **variable**, also called a **decision variable**, is an existentially quantified unknown of a problem.
- The **domain** of a decision variable  $x$ , here denoted by  $\text{dom}(x)$ , is the set of values in which  $x$  must take its value, if any.
- A **variable expression** takes a value that depends on the value of one or more decision variables.
- A **parameter** has a value from a problem description.
- Decision variables, parameters, and expressions are **typed**.

MiniZinc types are (arrays and sets of) Booleans, integers, floating-point numbers, enumerations, records, tuples, and strings, but not all these types can serve as types for decision variables.



# Decision Variables, Parameters, and Identifiers

---

- Decision variables and parameters in a model are concepts very different from programming variables in an imperative or object-oriented program.
- A decision variable in a model is like a variable in mathematics: it is *not* given a value in a model or a formula, and its value is only fixed in a solution, if a solution exists.
- A parameter in a model must be given a value, but only once: we say that it is **instantiated**.
- A decision variable or parameter is referred to by an **identifier**.
- An **index identifier** of an array **comprehension** takes on all its designated values in turn. Example: the index `c` in the 8-queens model.



# Parametric Models

---

- A parameter need not be instantiated inside a model.  
Example: drop “=8” from “`int: n=8`” in the 8-queens model to make it an `n`-queens model, and rename `8-queens.mzn` into `n-queens.mzn`.
- Data are values for parameters given outside a model:  
either in a datafile (.dzn suffix), or at the command line,  
or interactively in the integrated development environment (IDE).
- A parametric model has uninstantiated parameters.
- An instance is a pair of a parametric model and data.



# Modelling Concepts (end)

- A **constraint** is a restriction on the values that its decision variables can take together; equivalently, it is a Boolean-valued variable expression that must be true.
- An **objective function** is a numeric variable expression whose value is to be either minimised or maximised.
- An **objective** states what is being asked for:
  - find a first solution
  - find a solution minimising an objective function
  - find a solution maximising an objective function
  - find all solutions
  - count the number of solutions
  - prove that there is no solution
  - ...



# Constraint-Based Modelling

MiniZinc is a high-level **constraint-based** modelling language (*not* a solver):

- There are several **types** for decision variables: `bool`, `int`, `float`, `enum`, `string`, `tuple`, `record`, and `set`, possibly as elements of multidimensional matrices (`array`).
- There is a large vocabulary of **predicates** (`<`, `<=`, `=`, `!=`, `>=`, `>`, `all_different`, `circuit`, `regular`, ...), **functions** (`+`, `-`, `*`, `card`, `count`, `intersect`, `sum`, ...), and **logical connectives & quantifiers** (`not`, `/\`, `\/`, `->`, `<-`, `<->`, `forall`, `exists`, ...).
- There is support for *both* constraint **satisfaction** (`satisfy`) and constrained **optimisation** (`minimize` and `maximize`).

Most modelling languages are (much) lower-level than this!



# Correctness Is Not Enough for Models

Constraint  
Problems

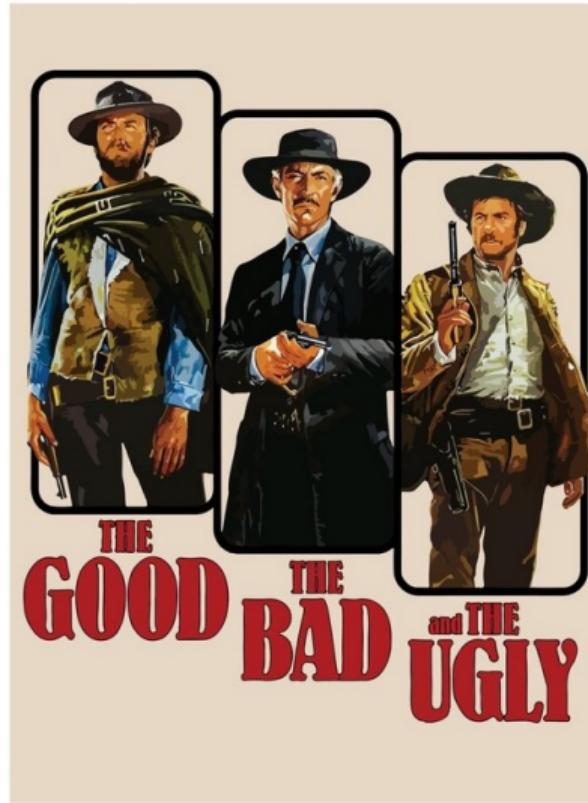
Combinatorial  
Optimisation

Modelling  
(in MiniZinc)

Solving

The MiniZinc  
Toolchain

Course  
Information





# Modelling is an Art!

There are good and bad models for each constraint problem:

- Different models of a problem may take different time on the same solver for the same instance.
- Different models of a problem may scale differently on the same solver for instances of growing size.
- Different solvers may take different time on the same model for the same instance.

Good modellers are worth their weight in gold!

Use solvers: based on decades of cutting-edge research, they are very hard to beat on exact solving.



# Outline

---

1. Constraint Problems

2. Combinatorial Optimisation

3. Modelling (in MiniZinc)

4. Solving

5. The MiniZinc Toolchain

6. Course Information



Solutions to a problem instance can be found by running a MiniZinc *backend*, that is a MiniZinc wrapper for a particular solver, on a file containing a model of the problem.

## Example (Solving the 8-queens instance)

Let us run the solver Gecode, of CP technology, from the command line:

```
minizinc --solver gecode 8-queens.mzn
```

The result is printed on stdout:

```
[4, 2, 7, 3, 6, 8, 5, 1]
```

This means that the queen of column 1 is in row 4 (note that MiniZinc uses 1-based indexing), the queen of column 2 is in row 2, and so on.

Use the command-line flag `-a` to ask for all solutions:

the line ----- is printed after each solution, but the

line ===== is printed after the last (the 92nd here) solution.



## Definition (Solving = Search + Reasoning + Relaxation)

- **Search**: Explore the space of candidate solutions.
- **Reasoning**: Reduce the space of candidate solutions.
- **Relaxation**: Exploit solutions to easier problems.

## Definition (Systematic Search: guarantees ultimately exact solving)

Progressively build a solution, and backtrack if necessary.

Use **reasoning** and **relaxation** in order to reduce the **search** effort.

It is used in most SAT, SMT, OMT, CP, LCG, and MIP solvers.

## Definition (Local Search: trades guarantee of exact solving for speed)

Start from a candidate solution and iteratively modify it a bit, until time-out.

It is the basic idea behind LS and genetic algorithm (GA) technologies.

For some details, see Topic 7: Solving Technologies.



# There Are So Many Solving Technologies

---

- No technology universally dominates all the others.
- One should test several technologies on each problem.
- Some technologies have **no** modelling languages:  
LS, DP, and GA are rather methodologies.
- Some technologies have **standardised** modelling languages across all solvers: SAT, SMT, OMT, and (M)IP.
- Some technologies have **non-standardised** modelling languages across their solvers: CP and LCG.



# Model and Solve

---

## Advantages:

- + Declarative model of a problem.
- + Easy adaptation to changing problem requirements.
- + Use of powerful solving technologies that are based on decades of cutting-edge research.

## Disadvantages:

- Do I need to learn several modelling languages? **No!**
- Do I need to understand the used solving technologies in order to get the most out of them? **Yes, but . . . !**



# Outline

---

1. Constraint Problems

2. Combinatorial Optimisation

3. Modelling (in MiniZinc)

4. Solving

5. The MiniZinc Toolchain

6. Course Information



# MiniZinc

MiniZinc is a declarative language (*not a solver*)  
for the constraint-based modelling of constraint problems:



- At Monash University, Australia
- Introduced in 2007; version 2.0 in 2014; version 3.0 imminent?
- Homepage: <https://www.minizinc.org>
- Integrated development environment (IDE)
- Annual **MiniZinc Challenge** for solvers, since 2008
- There are also **courses at Coursera**, also in Chinese



# MiniZinc Features

---

- Declarative language for modelling **what** the problem is
- Separation of problem **model** and instance **data**
- **Open-source** toolchain
- Much **higher**-level language than those of (M)IP and SAT
- Solver-**independent** language
- Solving-technology-**independent** language
- Vocabulary of **predefined** types, predicates and functions
- Support for **user-defined** predicates and functions
- Support for annotations with hints on **how** to solve
- Ever-growing number of users, solvers, and other tools



# MiniZinc Backends and Their Solvers

- SAT = Boolean satisfiability: Plingeling via PicatSAT, ...
- MIP = mixed integer programming: Cbc, FICO Xpress, Gurobi Optimizer, HiGHS, IBM ILOG CPLEX Optimizer, ...
- CP = constraint programming:  
Choco, Gecode, JaCoP, Mistral, SICStus Prolog, ...
- CBLS = constraint-based LS (local search), without exactness guarantee:  
Atlantis, OscaR.cbls via fzn-oscar-cbls, Yuck, ... : almost always time out
- LCG = lazy clause generation, a hybrid of CP and SAT: Chuffed, ...
- Other hybrid technologies: iZplus, MiniSAT(ID), SCIP, ...
- Portfolios: Google's CP-SAT of OR-Tools (with LCG, MIP, and LS), ...
- ..., SMT, OMT, ...



# MiniZinc Backends and Their Solvers

- SAT = Boolean satisfiability: **Plingeling** via **PicatSAT**, ...
- MIP = mixed integer programming: **Cbc**, FICO Xpress, **Gurobi Optimizer**, **HiGHS**, IBM ILOG CPLEX Optimizer, ...
- CP = constraint programming:  
Choco, **Gecode**, JaCoP, Mistral, SICStus Prolog, ...
- CBLS = constraint-based LS (local search), without exactness guarantee:  
Atlantis, OscaR.cbls via fzn-oscar-cbils, **Yuck**, ... : almost always time out
- LCG = lazy clause generation, a hybrid of CP and SAT: **Chuffed**, ...
- Other hybrid technologies: iZplus, MiniSAT(ID), SCIP, ...
- Portfolios: Google's **CP-SAT** of OR-Tools (with LCG, MIP, and LS), ...
- ..., SMT, OMT, ...

The backends installed on the IT department's ThinLinc hardware are in red.  
The commercial **Gurobi Optimizer** is under a free academic license:  
you may **not** use it for non-academic purposes.

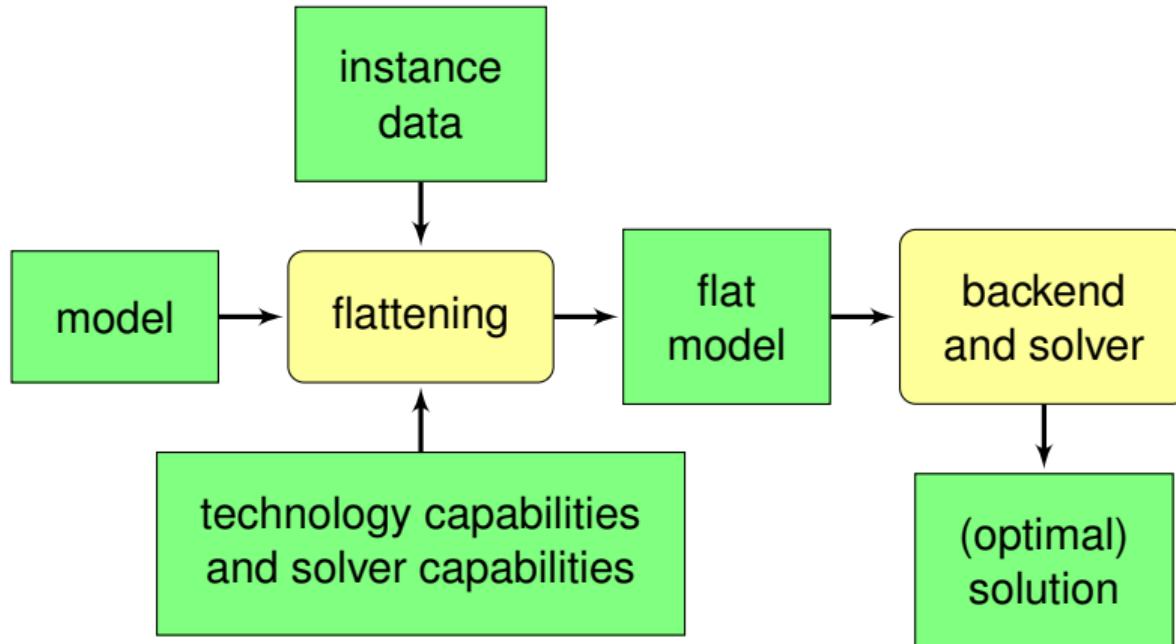


## MiniZinc Challenge 2024: 16 (of 20) Problems and Winners

Problem and Model	Backend and Solver	Solving Technology
aircraft-disassembly	CP-SAT (by Google)	portfolio: LCG, MIP, CBLS
cable-tree-wiring	CP-SAT (by Google)	portfolio: LCG, MIP, CBLS
community-detection	MZN/Gurobi	MIP
compression	PicatSAT	SAT
concert-hall-cap	MZN/Gurobi	MIP
fox-geese-corn	MZN/Gurobi	MIP
hoist-benchmark	Chuffed	hybrid: LCG = CP + SAT
monitor-placement-1id	CP Optimizer (by IBM)	CP
neighbours	CP-SAT (by Google)	portfolio: LCG, MIP, CBLS
peaceable_queens	Gecode-Dexter	portfolio: CP, LNS
portal	Chuffed	hybrid: LCG = CP + SAT
tiny-cvrp	MZN/Gurobi	MIP
train-scheduling	CP-SAT (by Google)	portfolio: LCG, MIP, CBLS
word-equations	PicatSAT	SAT
yumi-dynamic	CP-SAT (by Google)	portfolio: LCG, MIP, CBLS



# MiniZinc: Model Once, Solve Everywhere!



From a **single** language, one has access transparently to a wide range of solving technologies from which to choose.



# There Is No Need to Reinvent the Wheel!

Before solving, each decision variable of a **type** that is non-native to the targeted solver is replaced by decision variables of native types, using some well-known linear / clausal / ... encoding.

## Example (SAT)

The **order encoding** of integer decision variable `var 4..6: x` is

```
array[4..7] of var bool: B; % B[i] denotes truth of x >= i
constraint B[4];           % lower bound on x
constraint not B[7];        % upper bound on x
constraint B[4] \vee not B[5]; % consistency
constraint B[5] \vee not B[6]; % consistency
constraint B[6] \vee not B[7]; % consistency
```

For an integer decision variable with  $n$  domain values, there are  $n + 1$  Boolean decision variables and  $n$  clauses, all 2-ary.



Before solving, each use of a non-native **predicate** or **function** is replaced by:

- either: its MiniZinc-provided default definition,  
stated in terms of a kernel of imposed predicates;

## Example (default; not to be used for IP and MIP)

`all_different ([x, y, z])` gives  $x \neq y \wedge y \neq z \wedge z \neq x$ .

- or: a backend-provided solver-specific definition,  
using some well-known linear / clausal / ... encoding.

## Example (IP and MIP)

A compact linearisation of  $x \neq y$  is

```
var 0..1: p; % p = 1 denotes that x < y holds
int: Mx = ub(x-y+1); int: My = ub(y-x+1); % big-M constants
constraint x + 1 <= y + Mx * (1-p); % either x < y and p = 1
constraint y + 1 <= x + My * p; % or x > y and p = 0
```

One cannot naturally model graph colouring in IP,  
but the problem has integer decision variables (ranging over the colours).



# Benefits of Model-and-Solve with MiniZinc

---

- + Try many solvers of many technologies from 1 model.
- + A model improves with the state of the art of backends:
  - Type of decision variable: native representation or encoding.
  - Predicate: **reasoning**, **relaxation**, and definition.
  - Implementation of a solving technology.
- More on this in Topic 7: Solving Technologies.
- + For most managers, engineers, and scientists, it is easier with such a model-once-and-solve-everywhere toolchain to achieve good solution quality and high solving speed, including for harder data, and this without knowing (deeply) how the solvers work, compared to programming from first principles.



# How to Solve a Constraint Problem?

---

1 Model the problem

Constraint  
Problems

Combinatorial  
Optimisation

Modelling  
(in MiniZinc)

Solving

The MiniZinc  
Toolchain

Course  
Information

2 Solve the problem

Easy, right?



# How to Solve a Constraint Problem?

---

## 1 Model the problem

- Understand the problem
- Choose the decision variables and their domains
- Choose predicates to formulate the constraints
- Formulate the objective function, if any
- Make sure the model really represents the problem
- Iterate!

## 2 Solve the problem

- Choose a solving technology
- Choose a backend
- Choose a search strategy, if not black-box search
- Improve the model
- Run the model and interpret the (lack of) solution(s)
- Debug the model, if need be
- Iterate!

Easy, right?



# How to Solve a Constraint Problem?

## 1 Model the problem

- Understand the problem
- Choose the decision variables and their domains
- Choose predicates to formulate the constraints
- Formulate the objective function, if any
- Make sure the model really represents the problem
- Iterate!

## 2 Solve the problem

- Choose a solving technology
- Choose a backend
- Choose a search strategy, if not black-box search
- Improve the model
- Run the model and interpret the (lack of) solution(s)
- Debug the model, if need be
- Iterate!

Not so easy, but much easier than without a modelling tool!



# Outline

---

1. Constraint Problems

2. Combinatorial Optimisation

3. Modelling (in MiniZinc)

4. Solving

5. The MiniZinc Toolchain

6. Course Information



# Content

---

The use of tools for solving a combinatorial problem, by

- 1 first modelling it in a solving-technology-independent constraint-based modelling language, and
- 2 then running the model on an off-the-shelf solver.

We can now refine the course slogan:

How to solve combinatorial problems  
without knowing how to solve combinatorial problems?



# Learning Outcomes

---

In order to pass, the student must be able to:

- define the concept of combinatorial (optimisation or satisfaction) problem;
- explain the concept of constraint, as used in a constraint-based language;
- model a combinatorial problem in a solving-technology-independent constraint-based modelling language;
- compare empirically several models, say by introducing redundancy or by detecting and breaking symmetries;
- describe and compare solving technologies that can be used by the backends to a modelling language, including CP, LS, SAT, SMT, and MIP;
- choose suitable solving technologies for a new combinatorial problem, and motivate this choice;
- present and discuss topics related to the course content, orally and in writing, with a skill appropriate for the level of education.
  - ☞ written reports and oral resubmissions!



# Organisation and *Suggested Time Budget*

Period 2: early November to mid January, budget = 133.3 h:

- No textbook: slides, [MiniZinc documentation](#), [Coursera](#) courses
- 1 **warm-up session** for learning the [MiniZinc](#) toolchain
- 3 teacher-chosen **assignments** with 3 **help sessions**, 1 **grading session**, and 1 **solution session** each, to be done in student-chosen duo team:  
*suggested budget* = average of 21 hours / assignment / student (3 credits)
- 1 student-chosen **project**, with 8 **help sessions** (3 joint with Assignment 3), to be done in student-chosen duo team:  
*suggested budget* = 49.5 hours / student (2 credits)
- 12 **lectures**, including a **mandatory** guest lecture: budget = 21 hours
- Prerequisites: basic concepts in algebra, combinatorics, logic, graph theory, set theory, and implementation of basic search algorithms



# No Exam

---

Constraint  
Problems

Combinatorial  
Optimisation

Modelling  
(in MiniZinc)

Solving

The MiniZinc  
Toolchain

Course  
Information

The course has no exam!

You must demonstrate — by writing **reports** — that you cannot only correctly and efficiently solve a constraint problem via a model,

but also motivate and explain your model in terms of **all** the course concepts, and experimentally demonstrate the correctness and efficiency of your model.



# Lecture Topics

---

- Topic 1: Introduction
- Topic 2: Basic Modelling
- Topic 3: Constraint Predicates
- Topic 4: Modelling (for CP and LCG)
- Topic 5: Symmetry
- Topic 6: Case Studies
- Topic 7: Solving Technologies
- Topic 8: Reasoning & Search in CP & LCG
- (Topic 9: Modelling for CBLS)
- (Topic 10: Modelling for SAT, SMT, and OMT)
- (Topic 11: Modelling for MIP)



# 3 Assignment Cycles of 2 to 3 Weeks

---

Let  $D_i$  be the deadline day of Assignment  $i$ , with  $i \in 1..3$ :

- $D_i - 14$ : publication and all needed material was taught: start!
- $D_i - 8$ : help session a: participation strongly recommended!
- $D_i - 4$ : help session b: participation strongly recommended!
- $D_i - 2$ : help session c: participation strongly recommended!
- $D_i \pm 0$ : submission, by 13:00 Swedish time on a Friday
- $D_i + 5$  by 16:00: initial score  $a_i \in 0..5$  points
- $D_i + 6$ : teamwise oral grading session for some  $a_i \in \{1, 2\}$ : possibility of earning 1 extra point for final score; otherwise final score = initial score
- $D_i + 6 = D_{i+1} - 8$ : solution session and help session a



# Assignments (3 credits) and Overall Grade

The final score on Assignment 1 is actually “pass” or “fail”.

Let  $a_i \in 0..5$  be the **final score** on Assignment  $i$ , with  $i \in 2..3$ :

■ **20% threshold:**  $\forall i \in 2..3 : a_i \geq 20\% \cdot 5 = 1$

No catastrophic failure on individual assignments

■ **50% threshold:**  $a = a_2 + a_3 \geq 50\% \cdot (5 + 5) = 5$

The formula for the **assignment grade** in  $3..5$  is at the course homepage

■ **Worth going full-blast:** An **assignment sum**  $a \in 5..10$  is combined with a **project score**  $p \in 5..10$  in order to determine the **overall grade** in  $3..5$  according to a formula at the course homepage



# Project (2 credits)

## Topic:

- Model and solve a combinatorial problem that you are interested in, say for research, a course, a hobby, ...
- See the Project page at the course homepage for project ideas and the format for project proposals.

## Deadlines, inevitably overlapping with Assignments 2 and 3:

- Wed 19 Nov 2025 at 13:00: upload several project proposals
- Wed 26 Nov 2025 at 13:00: secure our approval; start!
- Fri 19 Dec 2025 at 13:00: upload initial project report
- Fri 16 Jan 2026 at 13:00: upload final project report; get score  $p \in 0..10$

**50% threshold:**  $p \geq 50\% \cdot 10 = 5$

The formula for the **project grade** in 3..5 is at the course homepage



# Project Guidelines

---

- Start early, despite the time overlap with Assignments 2 and 3.
- Attend the project help sessions, some jointly for Assignment 3.
- Read the Rules and Grading Criteria at the Project page.
- An **approach** is either a model for the entire problem, or a script (consider using **MiniZinc Python** and **JSON support**) with pre-processing + solving (possibly on a pipeline of multiple models) + post-processing: the final report is on **one** sufficiently complete and efficient approach.
- The initial report is on **one** approach, but it need be neither the final one, nor complete, nor efficient.



# Project Guidelines (end)

- Model the constraints incrementally, and be prepared to backtrack to the choice of decision variables (aka viewpoint).
- If the instances are too easy, then you still need to demonstrate skills in the advanced concepts (49.5h!).
- If the instances are too hard, then relax the problem (say by some loss of precision on the objective value) or some instances (or both).
- Collaborate with other teams that work on the same problem for the parsing, generation, or simplification of shared instances, and so on (but *not* for modelling). There is *no* competition between such teams.
- Consider also using the powerful local-search backend Gecode-LNS for the experiments (see Assignment 3).



# Assignment and Project Rules

Register a team by Sun 9 Nov 2025 at 23:59 at Studium:

- **Duo team:** Two consenting teammates sign up.
- **Solo team:** Apply in advance to the head teacher, who rarely agrees.
- **Random teammate?** Sign up at a Random Team; else you are bounced.

Other considerations:

- **Why (not) like this? Why no email reply?** See the FAQ list.
- **Teammate swapping:** Allowed, but to be declared to the helpdesk.
- **Teammate scores may differ** if no-show or passivity at grading session.
- **No freeloader:** Implicit honour declaration in reports that each teammate can individually explain everything; random checks will be made by us!
- **No plagiarism:** Implicit honour declaration in reports;  
extremely powerful detection tools will be used by us;  
suspected cases of using **or providing** must be reported!



# How To Communicate by Email or Studium?

- If you have a question about the **lecture material or course organisation**, then email the head teacher. An immediate answer will be given right before and after lectures, as well as during their breaks.
- If you have a question about the **assignments or infrastructure**, then contact the assistants at a help session or solution session for an immediate answer.

Short *clarification* questions (that is: *not* about modelling or programming issues) that are either emailed (see the address at the course website) or posted (at the Studium discussion) to the **M4CO helpdesk** are answered as soon as possible during working days and hours.

No answer means that you should go to a help session:  
**almost all the assistants' budgeted time is allocated to grading and to the help, grading, and solution sessions.**



# What Has Changed Since Last Time?

---

Changes made by the TekNat Faculty:

- The course is moved to period 2, which is the requisite 10 weeks long (unlike the previous, shorter period 1).

Changes triggered by the formal and informal course evaluations:

- There are skeleton reports specific to the project and each assignment.
- To reduce the workload, peer reviewing an initial project report is dropped.
- Two additional project-only help sessions are scheduled, between the deadlines of the initial and final project reports.



# What To Do Now?

---

- Bookmark and read the entire course website, especially its FAQ list.
- Read Sections 1 to 2.2 of the [MiniZinc Handbook](#).
- Get started on Assignment 1 and have questions ready for its first help session, which is already on Fri 7 Nov 2025.
- Register a duo team at Studium by Sun 9 Nov 2025 at 23:59, possibly upon advertising for a teammate at a course event or the discussion at Studium, and signing up at a Random Team at Studium as a last resort.
- Install the MiniZinc toolchain on your own hardware, if you have any.
- Be aware that few questions are tagged with MiniZinc at StackOverflow: you have to be prepared to read the documentation.