Topic 1: Introduction

(Version of 28th September 2022)

Pierre Flener and Jean-Noël Monette

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL442:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation

1Based partly on material by Guido Tack
Optimisation is a science of service: to scientists, to engineers, to artists, and to society.
## MiniZinc Challenge 2015: Some Winners

<table>
<thead>
<tr>
<th>Problem &amp; Model</th>
<th>Backend &amp; Solver</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Costas array</td>
<td>Mistral</td>
<td>CP</td>
</tr>
<tr>
<td>capacitated VRP</td>
<td>iZplus</td>
<td>hybrid</td>
</tr>
<tr>
<td>GFD schedule</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>grid colouring</td>
<td>MiniSAT(ID)</td>
<td>hybrid</td>
</tr>
<tr>
<td>instruction scheduling</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>large scheduling</td>
<td>Google OR-Tools.cp</td>
<td>CP</td>
</tr>
<tr>
<td>application mapping</td>
<td>JaCoP</td>
<td>CP</td>
</tr>
<tr>
<td>multi-knapsack</td>
<td>mzn-cplex</td>
<td>MIP</td>
</tr>
<tr>
<td>portfolio design</td>
<td>fzn-oscar-cbls</td>
<td>CBLS</td>
</tr>
<tr>
<td>open stacks</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>project planning</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>radiation</td>
<td>mzn-gurobi</td>
<td>MIP</td>
</tr>
<tr>
<td>satellite management</td>
<td>mzn-gurobi</td>
<td>MIP</td>
</tr>
<tr>
<td>time-dependent TSP</td>
<td>G12.FD</td>
<td>CP</td>
</tr>
<tr>
<td>zephyrus configuration</td>
<td>mzn-cplex</td>
<td>MIP</td>
</tr>
</tbody>
</table>
Outline

1. Constraint Problems
2. Combinatorial Optimisation
3. Modelling (in MiniZinc)
4. Solving
5. The MiniZinc Toolchain
6. Course Information
   Part 1: Modelling for Combinatorial Optimisation
   Part 2: Combinatorial Optimisation and CP
Contact
Outline

1. Constraint Problems
2. Combinatorial Optimisation
3. Modelling (in MiniZinc)
4. Solving
5. The MiniZinc Toolchain
6. Course Information
   Part 1: Modelling for Combinatorial Optimisation
   Part 2: Combinatorial Optimisation and CP
Example (Agricultural experiment design)

<table>
<thead>
<tr>
<th>plot1</th>
<th>plot2</th>
<th>plot3</th>
<th>plot4</th>
<th>plot5</th>
<th>plot6</th>
<th>plot7</th>
</tr>
</thead>
<tbody>
<tr>
<td>barley</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>corn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>millet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>oats</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rye</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>spelt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wheat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th></th>
<th>plot1</th>
<th>plot2</th>
<th>plot3</th>
<th>plot4</th>
<th>plot5</th>
<th>plot6</th>
<th>plot7</th>
</tr>
</thead>
<tbody>
<tr>
<td>barley</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>corn</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>millet</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>oats</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>rye</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>spelt</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>wheat</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th></th>
<th>Mon</th>
<th>Tue</th>
<th>Wed</th>
<th>Thu</th>
<th>Fri</th>
<th>Sat</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doctor A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doctor B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doctor C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doctor D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doctor E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Constraints** to be satisfied:

1. #on-call doctors / day = 1
2. #operating drs / weekday ≤ 2
3. #operating drs / week ≥ 7
4. #appointed drs / week ≥ 4
5. day off after operation day
6. . . .

**Objective function** to be minimised:

- Cost: . . .
### Example (Doctor rostering)

<table>
<thead>
<tr>
<th></th>
<th>Mon</th>
<th>Tue</th>
<th>Wed</th>
<th>Thu</th>
<th>Fri</th>
<th>Sat</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doctor A</td>
<td>call</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Doctor B</td>
<td>appt</td>
<td>call</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>none</td>
<td>call</td>
</tr>
<tr>
<td>Doctor C</td>
<td>oper</td>
<td>none</td>
<td>call</td>
<td>appt</td>
<td>appt</td>
<td>call</td>
<td>none</td>
</tr>
<tr>
<td>Doctor D</td>
<td>appt</td>
<td>oper</td>
<td>none</td>
<td>call</td>
<td>oper</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Doctor E</td>
<td>oper</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>call</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

**Constraints to be satisfied:**

1. `#on-call doctors / day = 1`
2. `#operating drs / weekday ≤ 2`
3. `#operating drs / week ≥ 7`
4. `#appointed drs / week ≥ 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

Demand vs capacity

Contingency planning

<table>
<thead>
<tr>
<th>Flow</th>
<th>Time Span</th>
<th>Hourly Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>From: Arlanda</td>
<td>00:00 – 09:00</td>
<td>3</td>
</tr>
<tr>
<td>To: west, south</td>
<td>09:00 – 18:00</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>18:00 – 24:00</td>
<td>2</td>
</tr>
<tr>
<td>From: Arlanda</td>
<td>00:00 – 12:00</td>
<td>4</td>
</tr>
<tr>
<td>To: east, north</td>
<td>12:00 – 24:00</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Airspace sectorisation

Workload balancing

Constraint Problems
Combinatorial Optimisation
Modelling (in MiniZinc)
Solving
The MiniZinc Toolchain
Course Information
Part 1: Modelling for Combinatorial Optimisation
Part 2: Combinatorial Optimisation and CP
Contact
COCP/M4CO 1
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 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

Haplotyping inference

Medical image analysis

Doctor rostering

Phylogenetic supertree

Haplotyping inference

Medical image analysis

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- & heterozygous sites:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C / G</th>
<th>T</th>
<th>C</th>
<th>A / T</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A / T</td>
<td>G</td>
<td>T</td>
<td>C / G</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>T</th>
<th>C</th>
<th>T</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>G</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>G</td>
<td>T</td>
<td>G</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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
Part 1: Modelling for Combinatorial Optimisation
Part 2: Combinatorial Optimisation and CP
Contact

Robot programming

Sensor-net configuration

Compiler design

Base-station testing

Applications in Programming and Testing

Robot programming

Sensor-net configuration

Compiler design

Base-station testing
Other Application Areas

### School timetabling

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:00</td>
<td>WFT202</td>
<td>LAMB13072</td>
<td>WFT202</td>
<td>LAMB13072</td>
</tr>
<tr>
<td>09:00</td>
<td>WFT202</td>
<td>LAMB13072</td>
<td>WFT202</td>
<td>LAMB13072</td>
</tr>
<tr>
<td>09:00</td>
<td>LAMB13072</td>
<td>WFT202</td>
<td>LAMB13072</td>
<td>WFT202</td>
</tr>
</tbody>
</table>

### Sports tournament design

### Security: SQL injection?

### Container packing
Definitions

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

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

Definitions

A candidate solution to a constraint problem maps each 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.
P \equiv NP 
(Cook, 1971; Levin, 1973)

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, or easy. Problems requiring super-polynomial time are considered intractable, or hard.
NP Completeness: Examples

Given a digraph \((V, E)\):

Examples

- Finding a **shortest path** takes \(\mathcal{O}(V \cdot E)\) time and is 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

- **2-SAT**: Determining the satisfiability of a conjunction of disjunctions of 2 Boolean literals is in P.
- **3-SAT**: Determining the satisfiability of a conjunction of disjunctions of 3 Boolean literals is NP-complete.
- **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:

<table>
<thead>
<tr>
<th>( n )</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>40 seconds</td>
</tr>
<tr>
<td>14</td>
<td>1 day</td>
</tr>
<tr>
<td>18</td>
<td>203 years</td>
</tr>
<tr>
<td>20</td>
<td>77k years</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>0.7 seconds</td>
</tr>
<tr>
<td>41</td>
<td>20 days</td>
</tr>
<tr>
<td>48</td>
<td>( 1.5 \cdot ) age of universe</td>
</tr>
</tbody>
</table>

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 it beats the \( \mathcal{O}(n!) \) algo on the 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!

The **Concorde TSP Solver** beats the Bellman-Held-Karp exact algo: it uses approximation & local-search algorithms, but it can sometimes prove the exactness (optimality) of its solutions. The largest instance it has solved exactly, in 136 CPU years in 2006, has 85,900 cities!  

☞ Let the fun begin!
Outline

1. Constraint Problems
2. Combinatorial Optimisation
3. Modelling (in MiniZinc)
4. Solving
5. The MiniZinc Toolchain
6. Course Information
   - Part 1: Modelling for Combinatorial Optimisation
   - Part 2: Combinatorial Optimisation and CP
Contact
A solving technology offers languages, methods, & tools for:

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

and / or

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

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

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

**Combinatorial** (= discrete) optimisation covers satisfaction and optimisation problems, for variables over discrete sets. 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&data as input:

- Boolean satisfiability (SAT)
- SAT (resp. optimisation) modulo theories (SMT & OMT)
- (Mixed) integer linear programming (IP & MIP)
- Constraint programming (CP)

... part 2 of 1DL442

- Hybrid technologies (LCG = CP + SAT, ...)

Methodologies, usually without modelling and solvers:

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

...
Outline

1. Constraint Problems
2. Combinatorial Optimisation
3. Modelling (in MiniZinc)
4. Solving
5. The MiniZinc Toolchain
6. Course Information

Part 1: Modelling for Combinatorial Optimisation
Part 2: Combinatorial Optimisation and CP
Contact
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:

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

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

Seen as a generate-and-test algorithm, it takes $O(n!)$ time, but it can be refined into the existing $O(n \log n)$ 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 (efficiently).
Modelling vs Programming

- Problem
- Specification
  - What? (declarative)
  - How? (imperative)
- Model
  - Automatic!
- Algorithm
  - Manual!
- Program

Modelling (in MiniZinc)
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)

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)

```
-2 array[1..9,1..9] of var 1..9: Sudoku;
-1
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)

<table>
<thead>
<tr>
<th></th>
<th>plot1</th>
<th>plot2</th>
<th>plot3</th>
<th>plot4</th>
<th>plot5</th>
<th>plot6</th>
<th>plot7</th>
</tr>
</thead>
<tbody>
<tr>
<td>barley</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>corn</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>millet</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>oats</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>rye</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>spelt</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>wheat</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th></th>
<th>plot1</th>
<th>plot2</th>
<th>plot3</th>
<th>plot4</th>
<th>plot5</th>
<th>plot6</th>
<th>plot7</th>
</tr>
</thead>
<tbody>
<tr>
<td>barley</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>corn</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>millet</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>oats</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>rye</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>spelt</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>wheat</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**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 \textbf{blocks} and the grains are \textbf{varieties}:

**Example (BIBD \textit{integer} model: \checkmark \rightsquigarrow 1 \text{ and } \rightsquigarrow 0)**

```plaintext
-3 enum Varieties; enum Blocks;
-2 int: blockSize; int: sampleSize; int: balance;
-1 array[Varieties,Blocks] of var 0..1: BIBD;
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)**

```plaintext
-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)**

```minizinc
enum Varieties; enum Blocks;
int: blockSize; int: sampleSize; int: balance;
array[Varieties,Blocks] of var 0..1: BIBD;
solve satisfy;
constraint forall(b in Blocks)
  (blockSize = count(BIBD[..,b], 1));
constraint forall(v in Varieties)
  (sampleSize = count(BIBD[v,..], 1));
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)**

```minizinc
Varieties = {barley,...,wheat}; Blocks = {plot1,...,plot7};
blockSize = 3; sampleSize = 3; balance = 1;
```
Using the `count` abstraction over linear expressions:

**Example (BIBD integer model: $\checkmark \sim 1$ and $-\sim 0$)**

```minizinc
enum Varieties; enum Blocks;
int: blockSize; int: sampleSize; int: balance;
array[Varieties,Blocks] of var 0..1: BIBD;
solve satisfy;
constraint forall(b in Blocks) (blockSize = count(BIBD[..,b], 1));
constraint forall(v in Varieties) (sampleSize = count(BIBD[v,..], 1));
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)**

```minizinc
Varieties = {barley,...,wheat}; Blocks = {plot1,...,plot7};
blockSize = 3; sampleSize = 3; balance = 1;
```
Reconsider the model fragment:

```plaintext
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)

<table>
<thead>
<tr>
<th>Grain</th>
<th>Plots</th>
</tr>
</thead>
<tbody>
<tr>
<td>barley</td>
<td>{plot1, plot2, plot3}</td>
</tr>
<tr>
<td>corn</td>
<td>{plot1, plot4, plot5}</td>
</tr>
<tr>
<td>millet</td>
<td>{plot1, plot6, plot7}</td>
</tr>
<tr>
<td>oats</td>
<td>{plot2, plot4, plot6}</td>
</tr>
<tr>
<td>rye</td>
<td>{plot2, plot5, plot7}</td>
</tr>
<tr>
<td>spelt</td>
<td>{plot3, plot4, plot7}</td>
</tr>
<tr>
<td>wheat</td>
<td>{plot3, plot5, plot6}</td>
</tr>
</tbody>
</table>

**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)

```plaintext
enum Varieties; enum Blocks;
int: blockSize; int: sampleSize; int: balance;
array[Varieties] of var set of Blocks: BIBD;
solve satisfy;
constraint forall(b in Blocks)
  (blockSize = sum(v in Varieties)(b in BIBD[v]));
constraint forall(v in Varieties)
  (sampleSize = card(BIBD[v]));
constraint forall(v, w in Varieties where v < w)
  (balance = card(BIBD[v] intersect BIBD[w]));
```
### Example (Doctor rostering)

<table>
<thead>
<tr>
<th>Doctor</th>
<th>Mon</th>
<th>Tue</th>
<th>Wed</th>
<th>Thu</th>
<th>Fri</th>
<th>Sat</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doctor A</td>
<td>call</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Doctor B</td>
<td>appt</td>
<td>call</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>none</td>
<td>call</td>
</tr>
<tr>
<td>Doctor C</td>
<td>oper</td>
<td>none</td>
<td>call</td>
<td>appt</td>
<td>appt</td>
<td>call</td>
<td>none</td>
</tr>
<tr>
<td>Doctor D</td>
<td>appt</td>
<td>oper</td>
<td>none</td>
<td>call</td>
<td>oper</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Doctor E</td>
<td>oper</td>
<td>none</td>
<td>oper</td>
<td>none</td>
<td>call</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

**Constraints to be satisfied:**

1. #on-call doctors / day = 1
2. #operating drs / weekday ≤ 2
3. #operating drs / week ≥ 7
4. #appointed drs / week ≥ 4
5. day off after operation day
6. ...

**Objective function to be minimised:**

- Cost: ...
constraint modelling for combinatorial optimisation

Example (Doctor rostering)

-4 set of int: Days; % d mod 7 = 1 iff d is a Monday
-3 enum Doctors;
-2 enum ShiftTypes = {appt, call, oper, none};
-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)

-4 Days = 1..7;
-3 Doctors = {Dr_A, Dr_B, Dr_C, Dr_D, Dr_E};
Using variables as indices within arrays: black magic?!

Example (Job allocation at minimal salary cost)

**Given** jobs $\text{Jobs}$ and the salaries of work applicants $\text{Apps}$, **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. $\text{array[Apps]}$ of $0..1000$: Salary;  % Salary[a]/job by a
2. $\text{array[Jobs]}$ of var Apps: Worker;  % job j by Worker[j]
3. solve minimize $\text{sum}(j \text{ in Jobs})(\text{Salary[Worker[j]])};$
4. constraint ...;  % qualifications, workload, etc
Using variables as indices within arrays: **black magic?!**

### Example (Vehicle routing: backbone model)

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

<table>
<thead>
<tr>
<th></th>
<th>AMS</th>
<th>BRU</th>
<th>LUX</th>
<th>CDG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```plaintext
array[Cities, Cities] of float: Dist; % instance data

array[Cities] of var Cities: Next;  % from c to Next[c]

solve minimize sum(c in Cities)(Dist[c, Next[c]]);

constraint circuit(Next);

constraint ...; % side constraints, if any
```
Using variables as indices within arrays: **black magic?!**

**Example (Vehicle routing: backbone model)**

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

AMS    BRU    LUX    CDG
Next:   BRU    AMS    CDG    LUX

So `all_different(Next)` is too weak!
```
Using variables as indices within arrays: black magic?!

Example (Vehicle routing: backbone model)

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

AMS  BRU  LUX  CDG
Next:  BRU  CDG  AMS  LUX

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

![Diagram of vehicle routing model with cities and distances]

Using 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: Dist; % instance data
2 array[Cities] of var Cities: Next;% from c to Next[c]
3 solve minimize sum(c in Cities)(Dist[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 variable per queen:

Let variable \( \text{Row}[c] \), of domain 1..8, represent the row of the queen in column \( c \), for \( c \) in a..h, renamed into 1..8.

Example: \( \text{Row}[3] = 4 \) means the queen of column 3 is in row 4. The constraint that all queens 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 variables \( \text{Row}[c] \) 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`:

```plaintext
% Model of the 8-queens problem
include "globals.mzn";
% parameter:
int: n = 8; % n denotes the given number of queens
% Row[c] denotes the row of the queen in column c:
array[1..n] of var 1..n: Row; % variables and domains
% constraints:
constraint all_different( Row );
constraint all_different([Row[c]+c | c in 1..n]);
constraint all_different([Row[c]-c | c in 1..n]);
% objective:
solve satisfy; % solve to satisfaction
% pretty-printing of solutions:
output [show(Row)];
```

The `all_different` constraint predicate requires that all its argument expressions take different values.
Modelling Concepts

- A **variable**, also called a **decision variable**, is an existentially quantified unknown of a problem.

- The **domain** of a 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.

- Variables, parameters, and expressions are **typed**.

MiniZinc types are (arrays and sets of) Booleans, integers, floating-point numbers, enumerations, and strings, but not all these types can serve as types for variables.
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 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 variable or parameter is referred to by an identifier.

- An index identifier of an array comprehension takes on all its possible values in turn.

Example: the index $c$ in the 8-queens model.
Parametric Models

- A parameter need not be instantiated inside a model. Ex: drop "=8" from "int: n=8" in the 8-queens model in order to make it an n-queens model, and rename the file 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 variables can take conjointly; 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 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
  - ...
MiniZinc is a high-level constraint-based modelling language (not a solver):

- There are several **types** for variables: `int`, `enum`, `float`, `bool`, `string`, and `set`, possibly as elements of multidimensional matrices (`array`).

- There is a nice vocabulary of **predicates** (`<`, `<=`, `=`, `!=`, `>=`, `>`, `all_different`, `circuit`, `regular`, ...), **functions** (`+`, `−`, `∗`, `card`, `count`, `intersect`, `sum`, ...), and **connectives** (`not`, `/\`, `\`, `→`, `<-, <->`, ...).

- 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
Modelling is an Art!

There are good & 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
   Part 1: Modelling for Combinatorial Optimisation
   Part 2: Combinatorial Optimisation and CP
   Contact
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, 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 (92nd) solution.
How Do Solvers Work?

Definition (Solving = Search + Inference + Relaxation)

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

Definition (Systematic Search)

Progressively build a solution, and backtrack if necessary. Use **inference** and **relaxation** to reduce the search effort. It is used in most SAT, SMT, OMT, CP, LCG, & MIP solvers.

Definition (Local Search)

Start from a candidate solution and iteratively modify it. It is the basic idea behind LS and GA technologies.

For 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, & (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:

– Need to learn several modelling languages? No!

– 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
   Part 1: Modelling for Combinatorial Optimisation
   Part 2: Combinatorial Optimisation and CP
   Contact
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
- 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 & SAT
- Solver-independent language
- Solving-technology-independent language
- Vocabulary of predefined types, predicates & 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, ...
- OMT = optimisation modulo theories:
  OptiMathSAT via emzn2fzn + fzn2omt
- MIP = mixed integer programming: Cbc, FICO Xpress, Gurobi Optimizer, IBM ILOG CPLEX Optimizer, ...
- CP = constraint programming:
  Choco, Gecode, JaCoP, Mistral, SICStus Prolog, ...
- CBLS = constraint-based LS (local search):
  OscaR.cbls via fzn-oscar-cbls, Yuck, ...
- LCG = lazy clause generation, a hybrid of CP and SAT:
  Chuffed, Google’s CP-SAT of OR-Tools, ...
- Other hybrid technos: iZplus, MiniSAT(ID), SCIP, ...
- Portfolios of solvers: sunny-cp, ...
MiniZinc Backends and Their Solvers

- SAT = Boolean satisfiability: Plingeling via PicatSAT, ...
- OMT = optimisation modulo theories: OptiMathSAT via emzn2fzn + fzn2omt
- MIP = mixed integer programming: Cbc, FICO Xpress, Gurobi Optimizer, IBM ILOG CPLEX Optimizer, ...
- CP = constraint programming: Choco, Gecode, JaCoP, Mistral, SICStus Prolog, ...
- CBLS = constraint-based LS (local search): OscaR.cbls via fzn-oscar-cbls, Yuck, ...
- LCG = lazy clause generation, a hybrid of CP and SAT: Chuffed, Google’s CP-SAT of OR-Tools, ...
- Other hybrid technos: iZplus, MiniSAT(ID), SCIP, ...
- Portfolios of solvers: sunny-cp, ...

Backends installed on IT dept’s ThinLinc hardware are red. The commercial Gurobi Optimizer is under a free academic license: you may not use it for non-academic purposes.
## MiniZinc Challenge 2015: Some Winners

<table>
<thead>
<tr>
<th>Problem &amp; Model</th>
<th>Backend &amp; Solver</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Costas array</td>
<td>Mistral</td>
<td>CP</td>
</tr>
<tr>
<td>capacitated VRP</td>
<td>iZplus</td>
<td>hybrid</td>
</tr>
<tr>
<td>GFD schedule</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>grid colouring</td>
<td>MiniSAT(ID)</td>
<td>hybrid</td>
</tr>
<tr>
<td>instruction scheduling</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>large scheduling</td>
<td>Google OR-Tools.cp</td>
<td>CP</td>
</tr>
<tr>
<td>application mapping</td>
<td>JaCoP</td>
<td>CP</td>
</tr>
<tr>
<td>multi-knapsack</td>
<td>mzn-cplex</td>
<td>MIP</td>
</tr>
<tr>
<td>portfolio design</td>
<td>fzn-oscar-cbls</td>
<td>CBLS</td>
</tr>
<tr>
<td>open stacks</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>project planning</td>
<td>Chuffed</td>
<td>LCG</td>
</tr>
<tr>
<td>radiation</td>
<td>mzn-gurobi</td>
<td>MIP</td>
</tr>
<tr>
<td>satellite management</td>
<td>mzn-gurobi</td>
<td>MIP</td>
</tr>
<tr>
<td>time-dependent TSP</td>
<td>G12.FD</td>
<td>CP</td>
</tr>
<tr>
<td>zephyrus configuration</td>
<td>mzn-cplex</td>
<td>MIP</td>
</tr>
</tbody>
</table>

(portfolio and parallel categories omitted)
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 variable of a **type** that is non-native to the targeted solver is replaced by variables of native types, using some well-known linear / clausal / . . . encoding.

### Example (SAT)

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

```plaintext
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] \/ not B[5]; % consistency
constraint B[5] \/ not B[6]; % consistency
constraint B[6] \/ not B[7]; % consistency
```

For an integer variable with \( n \) domain values, there are \( n + 1 \) Boolean 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!=y/\y!=z/\z!=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

```plaintext
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 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:
  • Variable type: native representation or encoding.
  • Predicate: inference, 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-&-solve-everywhere toolchain to achieve good solution quality and high solving speed, including for harder and bigger data, and without knowing (deeply) how the solvers work, compared to programming from first principles.
How to Solve a Constraint Problem?

1. Model the problem
   - Understand the problem
   - Choose the decision variables and their domains
   - Choose predicates to model the constraints
   - Model 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 model the constraints
   - Model 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 model the constraints
   • Model 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
   Part 1: Modelling for Combinatorial Optimisation
   Part 2: Combinatorial Optimisation and CP
   Contact
Outline

1. Constraint Problems
2. Combinatorial Optimisation
3. Modelling (in MiniZinc)
4. Solving
5. The MiniZinc Toolchain
6. Course Information
   Part 1: Modelling for Combinatorial Optimisation
   Part 2: Combinatorial Optimisation and CP
   Contact
Content of Part 1 = M4CO (course 1DL451)

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.
Learning Outcomes of Part 1 = M4CO

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

- define the concept of combinatorial problem;
- explain the concept of constraint, as used in a constraint-based modelling language;
- model a combinatorial problem in a constraint-based solving-technology-independent 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 constraint-based modelling language, including CP, LS, SAT, SMT, and MIP;
- choose suitable solving technologies for a new combinatorial problem, and motivate this choice;
- present & discuss topics related to the course content, orally and in writing, with a skill appropriate for the level of education. ✉️ written reports & oral resubmissions!
Organisation and Time Budget of Part 1

Period 1: late August to late October, budget = 133.3 h:

- No textbook: slides, MiniZinc documentation, Coursera
- 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: budget = avg 22 hours/assignment/student (3 credits)
- 1 student-chosen project, to be done in student-chosen duo team, and peer review of other team’s initial report: budget = 45 hours/student (2 credits)
- 12 lectures, including a mandatory guest lecture, plus 3 mandatory project presentation sessions: budget = 22.5 hours
- Prerequisites: basic algebra, combinatorics, logic, graph theory, set theory, and search algorithms
No Exams in Part 1 and Part 2

Both M4CO (1DL451) and COCP (1DL442) have no exam!

You must demonstrate — by writing reports and making an oral project presentation — that you cannot only code, namely:

- correctly & efficiently solve a constraint problem via a model (in Part 1), and

- design a correct & efficient inference algorithm or search algorithm for a CP solver (in Part 2),

but also motivate and explain your code in terms of all the concepts of the lectures, as well as experimentally demonstrate the correctness & efficiency of your code.
Lecture Topics of Part 1 = M4CO

- Topic 1: Introduction
- Topic 2: Basic Modelling
- Topic 3: Constraint Predicates
- Topic 4: Modelling (for CP & LCG)
- Topic 5: Symmetry
- Topic 6: Case Studies
- Topic 7: Solving Technologies
- Topic 8: Inference & 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 2–3 Weeks in Part 1

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

- $D_i - 14$: publication & all needed material taught: start!
- $D_i - 8$: help session a: participation recommended!
- $D_i - 4$: help session b: participation recommended!
- $D_i - 2$: help session c: participation 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 & help session a
Assignments (3 c) & Overall Grade in Part 1

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

Let $a_i \in 0..5$ be 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:** $m = a_2 + a_3 \geq 50\% \cdot (5 + 5) = 5$
  
  ☞ the formulae for the modelling assignment grade and project grade in 3..5 are at the course homepage

- **Worth going full-blast:** A modelling assignment sum $m \in 5..10$ is combined with a project score $p \in 5..10$ in order to determine the overall grade in 3..5 for 1DL451 according to a formula at the course homepage
Project (2 credits) in Part 1 = M4CO

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 ideas for projects and the format for a project proposal.

Deadlines in 2022 (overlap with Assignments 2 and 3):

- Wed 14 Sep at 13:00: upload proposal
- Wed 21 Sep at 13:00: secure our approval; start!
- Fri 14 Oct at 13:00: upload initial report
- Mon 17 & Tue 18 Oct: present, oppose, upload slides
- Wed 19 Oct at 13:00: upload peer review
- Fri 28 Oct at 13:00: upload final report; score $p \in 0..10$

The length & order of presentations will be fixed in due time.
Project Guidelines

- Start early, despite overlap with Assignments 2 and 3.
- Attend the help sessions (some with Assignment 3).
- Read the Rules & Grading Criteria at the Project page.
- An approach is either a model for the entire problem, or a script (consider using MiniZinc-Python) with pre-processing, solving (possibly on a pipeline of multiple models), and post-processing: the final report is on one sufficiently complete and efficient approach.
- The initial report & presentation are on one approach, but it need be neither the final one, nor complete.
Model the constraints incrementally, and be prepared to backtrack to the choice of variables (aka viewpoint).

If the instances are too easy, then you still need to demonstrate skills in the advanced concepts (45h!).

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 4 Sep 2022 at 23:59 at Studium:

- **Duo team:** Two consenting partners sign up
- **Solo team:** Apply to head teacher, who rarely agrees
- **Random partner?** Assent to TAs, else you’re bounced

Other considerations:

- **Why (not) like this? Why no email reply?** See FAQ
- **Partner swapping:** Allowed, but to be declared to TAs
- **Partner scores may differ** if no-show or passivity
- **No freeloader:** Implicit honour declaration in reports that each partner 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 will be reported
Outline

1. Constraint Problems
2. Combinatorial Optimisation
3. Modelling (in MiniZinc)
4. Solving
5. The MiniZinc Toolchain
6. Course Information
   Part 1: Modelling for Combinatorial Optimisation
   Part 2: Combinatorial Optimisation and CP
Contact
Learning Outcomes of Part 2 = CO and CP

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

- describe how a CP solver works, by giving its architecture and explaining the principles it is based on;
- augment a CP solver with a propagator for a new constraint predicate, and evaluate empirically whether the propagator is better than a definition based on the existing constraint predicates of the solver;
- devise empirically a (problem-specific) search strategy that can be used by a CP solver;
- design and compare empirically several constraint programs (with model and search parts) for a combinatorial problem;
- present & discuss topics related to the course content, orally and in writing, with a skill appropriate for the level of education. ✝ written reports!
Organisation and Time Budget of Part 2

Period 2: late October to mid January(!), budget = 133.3 h:

- 12 lectures, including a mandatory guest lecture: budget = 19.5 hours

- No textbook: slides and MiniCP teaching materials

- 1 warm-up session for learning the MiniCP toolchain

- 3 teacher-chosen assignments, with 3 help sessions and 1 solution session each (but no grading session), to be done in student-chosen duo team: budget = avg 38 hours / assignment / student (5 credits)

- Prerequisites: Java; basic algebra, combinatorics, logic, graph theory, set theory, and search algorithms
Lecture Topics of Part 2

- Topic 12: CP and the MiniCP Solver
- Part 1: CP, Filtering, Search, Consistency, Fixpoint
- Part 2: Domains, Variables, Constraints
- Part 3: Memory Management (Trail + Copy) & Search
- Part 4: Sum and Element Constraints
- Part 5: Circuit Constraint, TSP, and LNS
- Part 6: AllDifferent Constraint
- Part 7: Table Constraints
- Part 8: Search
- Part 9: Cumulative Scheduling
- Part 10: Disjunctive Scheduling
- Topic 18: Conclusion
Let $D_i$ be the deadline day of Assignment $i$, with $i \in 4..6$:

- $D_i - 14$: publication & all needed material taught: start!
- $D_i - 7$: help session a: participation recommended!
- $D_i - 4$: help session b: participation recommended!
- $D_i - 2$: help session c: participation recommended!
- $D_i \pm 0$: submission, by 13:00 Swedish time on a Friday
- $D_i + 6$ by 16:00: final score $a_i \in 0..5$ points
- No initial grade and no grading session!
- $D_i + 6 = D_{i+1} - 8$: solution session & help session a
Assignments (5 c) in Part 2 & Overall Grade

The final score on Assignment 4 is actually “pass” or “fail”. Let $a_i \in 0..5$ be final score on Assignment $i$, with $i \in 5..6$:

- **20% threshold:** $\forall i \in 5..6 : a_i \geq 20\% \cdot 5 = 1$
  No catastrophic failure on individual assignments

- **50% threshold:** $c = a_5 + a_6 \geq \lceil 50\% \cdot (5 + 5) \rceil = 5$
 ☞ the formula for the programming assignment grade in 3..5 is at the course homepage

- **Worth going full-blast:** A modelling assignment sum $m \in 5..10$ is combined with a project score $p \in 5..10$ and a programming assignment sum $c \in 5..10$ in order to determine the overall grade in 3..5 for 1DL442 according to a formula at the course homepage
Assignment Rules

Register new team by Sun 6 Nov 2022 at 23:59 by email:

- **Duo team**: Two consenting partners write to TAs
- **Solo team**: Apply to head teacher, who rarely agrees

Other considerations:

- **Why (not) like this? Why no email reply?** See FAQ
- **Partner swapping**: Allowed, but to be declared to TAs
- **Partner scores may differ** if no-show or passivity
- **No freeloader**: Implicit honour declaration in reports that each partner 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 will be reported
Outline

1. Constraint Problems
2. Combinatorial Optimisation
3. Modelling (in MiniZinc)
4. Solving
5. The MiniZinc Toolchain
6. Course Information
   Part 1: Modelling for Combinatorial Optimisation
   Part 2: Combinatorial Optimisation and CP
Contact
If you have a question about the lecture material or course organisation, then contact 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 emailed (see address at course website) or posted (at Studium discussion) to the COCP 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?

Change made by the TekNat Faculty:
- Period 1 lasts 9 weeks (not 8.5): more time for Project after Assignment 3 (but still work on them in parallel).

Changes wanted by the head teacher and assistants:
- No videos, no Zoom: participation still recommended.
- Even more automation is in our script for experiments.
- The timetable of Part 2 is adjusted upon our first experience with the MiniCP material in autumn 2021.

Changes triggered by the course evaluations:
- The help sessions a & b for Assignment 1 are 24h later.
- 6 (not 5) help sessions are scheduled for the project.
- The modelling checklists are available separately.
- A slide on the debugging of models is added.
What To Do Now in Part 1?

- Bookmark & read course website, especially FAQs.
- 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 on Thu 1 Sep 2022.
- Register a duo team by Sun 4 Sep 2022 at 23:59, possibly upon advertising for a teammate at a course event or the discussion at Studium, and requesting a random teammate from the helpdesk as a last resort.
- Install the MiniZinc toolchain on your hardware, if any.
- Be aware that few questions are tagged with MiniZinc at StackOverflow: you have to read the documentation.
What To Do Now in Part 2?

- Bookmark & re-read course website, especially FAQs.
- Get started on Assignment 4 and have questions ready for the first help session, which is on Fri 11 Nov 2022.
- Inform us of a new duo team by Sun 6 Nov at 23:59, possibly upon advertising for a teammate at a course event or the discussion at Studium, and requesting a random teammate from the helpdesk as a last resort.
- Install MiniCP on your hardware, if any.
- Be aware that there is no StackOverflow-like website for avoiding to have to read the MiniCP documentation.