Topic 1: Introduction
(Version of 11th October 2017)

Pierre Flener and Jean-Noël Monette

ASTRA Research Group on Combinatorial Optimisation
Uppsala University Sweden

Course 1DL448: Modelling for Combinatorial Optimisation

1Based also on some 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</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>MZN / 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
Outline

1. Constraint Problems

2. Combinatorial Optimisation

3. Modelling (in MiniZinc)

4. Solving

5. The MiniZinc Toolchain

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

**Instance data:** 7 rows, 7 cols, 3 grains, 3 plots, 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 sample size:** Every grain is grown in 3 plots.
2. **Equal growth load:** Every plot grows 3 grains.
3. **Balance:** Every grain pair is grown in 1 common plot.

**Instance data:** 7 rows, 7 cols, 3 grains, 3 plots, 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. #doctors-on-call / day = 1
2. #operations / workday ≤ 2
3. #operations / week ≥ 7
4. #appointments / week ≥ 4
5. day off after operation day
6. ...

**Objective function to be minimised:**

- Cost: ...
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>A</td>
<td>call</td>
<td>–</td>
<td>oper</td>
<td>–</td>
<td>oper</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>B</td>
<td>app</td>
<td>call</td>
<td>–</td>
<td>oper</td>
<td>–</td>
<td>–</td>
<td>call</td>
</tr>
<tr>
<td>C</td>
<td>oper</td>
<td>–</td>
<td>call</td>
<td>app</td>
<td>app</td>
<td>call</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>app</td>
<td>oper</td>
<td>–</td>
<td>call</td>
<td>oper</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>E</td>
<td>oper</td>
<td>–</td>
<td>oper</td>
<td>–</td>
<td>call</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Constraints to be satisfied:
1. \#doctors-on-call / day = 1
2. \#operations / workday ≤ 2
3. \#operations / week ≥ 7
4. \#appointments / 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

Airspace sectorisation

Contingency planning

<table>
<thead>
<tr>
<th>Flow</th>
<th>Time Span</th>
<th>Hourly Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>From: Arland</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: Arland</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>

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, and a targeted number $s$ of sectors.

**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

Haplotype inference

Medical image analysis

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

```
Oceanodroma castro
   Hydrobates pelagicus
   Macronectes giganteus
   Fulmarus glacialoides
   Fulmarus glacialis
   Bulweria bulwerii
   Procellaria cinerea
   Calonectris diomedea
   Puffinus assimilis
   Puffinus puffinus
   Puffinus yelkouan
   Puffinus mauretanicus
   THALASSARCHE BULLERI
   Thalassarche chrysostoma
   Phoebetria fusca
   Phoebetria palpebrata
   Phoebastria albatrus
   Phoebastria immutabilis
   Diomedea amsterdamensis
   DIOMEDEA EPOMOPHORA

Pygoscelis adeliae
   Eudyptula minor
   Megadyptes antipodes
   Eudyptes pachyrhynchus
   Pelagodroma marina
   DIOMEDEA EPOMOPHORA
   THALASSARCHE BULLERI
   Daption capense
   Pelecanoides georgicus
   Pachyptila vittata
   Pachyptila turtur
   Procellaria westlandica
   Puffinus griseus
   Puffinus huttoni
   Pterodroma inexpectata
   Pterodroma cookii
```
Example (Haplotype inference by pure parsimony)

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

- · · ·
  - A | C / G |
  - C | A / T |
- · · ·
  - 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

Robot task sequencing

Sensor net configuration

Compiler design

Base station testing

Robot task sequencing

Sensor net configuration

Compiler design

Base station testing
Other Application Areas

### School timetabling

<table>
<thead>
<tr>
<th>Time</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:00</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
</tr>
<tr>
<td>9:00</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
</tr>
<tr>
<td>10:00</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
</tr>
<tr>
<td>11:00</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
</tr>
<tr>
<td>12:00</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
</tr>
<tr>
<td>13:00</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
</tr>
<tr>
<td>14:00</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
<td>LAB2020</td>
<td>INF2021</td>
</tr>
</tbody>
</table>

### Sports tournament design

- **School timetabling**
- **Sports tournament design**

### Security: SQL injection

### Container packing
Definition

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.

Definition

A candidate solution to a constraint problem assigns to each variable 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$. If the answer is ‘yes’, then the other six problems can be settled by a computer.

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** ≠ **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*. 

\[ P \quad ? \quad NP \quad \text{(Cook, 1971; Levin, 1973)} \]
NP Completeness: Examples

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

**Example**

- Finding a **shortest path** takes \(O(V \cdot E)\) time and is in P.
- Determining the existence of a simple path (which has distinct vertices), from a given single source, that has \textit{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’.

**Example**

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

Example

- **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 and can only be solved exactly & 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 $O(n^2 \cdot 2^n)$ 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 $O(n!)$ 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
A solving technology offers methods and 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 software that takes a model as input and tries to solve the modelled problem.

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 a model as input:
- Boolean satisfiability (SAT)
- SAT modulo theories (SMT)
- (Mixed) integer linear programming (IP and MIP)
- Constraint programming (CP)
- . . .
- 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
Modelling vs Programming

- problem
  - specification
    - what? (declarative)
    - how? (imperative)
      - model
        - automatic!
      - algorithm
        - manual!
      - program
  - program
### 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 sample size:** Every grain is grown in 3 plots.
2. **Equal growth load:** Every plot grows 3 grains.
3. **Balance:** Every grain pair is grown in 1 common plot.

**Instance data:** 7 rows, 7 cols, 3 grains, 3 plots, 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 sample size: Every grain is grown in 3 plots.
2. Equal growth load: Every plot grows 3 grains.
3. Balance: Every grain pair is grown in 1 common plot.

**Instance data**: 7 rows, 7 cols, 3 grains, 3 plots, balance 1. General term: **balanced incomplete block design (BIBD)**.
Example (BIBD integer model: ✓ → 1 and – → 0)

1 int: nbrVarieties; int: nbrBlocks;
2 set of int: Varieties = 1..nbrVarieties;
3 set of int: Blocks = 1..nbrBlocks;
4 int: sampleSize; int: blockSize; int: balance;
5 array[Varieties,Blocks] of var 0..1: BIBD;
6 solve satisfy;
7 constraint forall(v in Varieties)
8  (sampleSize = count(BIBD[v,..],1)); % no slicing yet
9 constraint forall(b in Blocks)
10  (blockSize = count(BIBD[..,b],1));
11 constraint forall(v, w in Varieties where v < w)
12  (balance=count([BIBD[v,b]*BIBD[w,b]|b in Blocks],1));

Example (Instance data for our AED)

1 nbrVarieties = 7; nbrBlocks = 7;
2 sampleSize = 3; blockSize = 3; balance = 1;
Reconsider the model fragment:

```minizinc
constraint forall(v, w in Varieties where v < w)
(balance = count([BIBD[v,b] * BIBD[w,b] | b in Blocks], 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 two ordered varieties* \( v \) *and* \( w \),

*the count of blocks* \( b \) *whose product* \( BIBD[v,b] \ast BIBD[w,b] \) *is* 1

*must equal* balance

The constraint is **not procedural**:?

*for all two ordered varieties* \( v \) *and* \( w \),

*we first count the blocks* \( b \) *whose product* \( BIBD[v,b] \ast BIBD[w,b] \) *is* 1,

*and then we check if that count equals* balance

The latter reading is appropriate for solution **checking**, but solution **finding** performs no such procedural summation.
Example (Idea for another BIBD model)

<table>
<thead>
<tr>
<th>Grain</th>
<th>Sets</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 sample size: Every grain is grown in 3 plots.
2. Equal growth load: Every plot grows 3 grains.
3. Balance: Every grain pair is grown in 1 common plot.
Example (BIBD set model: a block set per variety)

```
array[Varieties] of var set of Blocks: BIBD;

(sampleSize = card(BIBD[v]));

(blockSize = count(BIBD,b)); % syntax fixed at T02

(balance = card(BIBD[v] inter BIBD[w]));
```

Example (Instance data for our AED)

```
...  
...  
```
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:**
- #doctors-on-call / day = 1
- #operations / workday ≤ 2
- #operations / week ≥ 7
- #appointments / week ≥ 4
- day off after operation day
- . . .

**Objective function to be minimised:**
- Cost: . . .
### 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>A</td>
<td>call</td>
<td>–</td>
<td>oper</td>
<td>–</td>
<td>oper</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>B</td>
<td>app</td>
<td>call</td>
<td>–</td>
<td>oper</td>
<td>–</td>
<td>–</td>
<td>call</td>
</tr>
<tr>
<td>C</td>
<td>oper</td>
<td>–</td>
<td>call</td>
<td>app</td>
<td>app</td>
<td>call</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>app</td>
<td>oper</td>
<td>–</td>
<td>call</td>
<td>oper</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>E</td>
<td>oper</td>
<td>–</td>
<td>oper</td>
<td>–</td>
<td>call</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

**Constraints to be satisfied:**
- \#doctors-on-call / day \(= 1\)
- \#operations / workday \(\leq 2\)
- \#operations / week \(\geq 7\)
- \#appointments / week \(\geq 4\)
- day off after operation day
- . . .

**Objective function to be minimised:**
- Cost: . . .
Example (Doctor rostering, in future MiniZinc syntax)

```mini
set of int: Days = 1..7;
set of int: Mon2Fri = 1..5;
enum Doctors = {Dr_A, Dr_B, Dr_C, Dr_D, Dr_E};
enum ShiftTypes = {app, call, oper, none};

array[Doctors,Days] of var ShiftTypes: Roster;

solve minimize ...; % plug in an objective function

constraint forall(d in Days)
  (count(Roster[..,d], call) = 1);
constraint forall(w in Mon2Fri)
  (count(Roster[..,w], oper) <= 2);
constraint count(Roster, oper) >= 7;
constraint count(Roster, app) >= 4;
constraint forall(d in Doctors)
  (regular(Roster[d,..], (oper none|app|call|none)*));
... % other constraints
```
Example (Sudoku model, in future MiniZinc syntax)

```mini
array[1..9,1..9] of var 1..9: Sudoku;

% load the hints
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 {1,4,7}) (alldifferent(Sudoku[i..i+2,j..j+2]));
```

This solves the hardest Sudokus in less than a millisecond!
Using variables as indices: black magic?!

Example (Job allocation at minimal salary cost)

Given \( n \) jobs and the salaries of \( w \) work applicants, 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. \( \text{array}[1..w] \) of \( \text{int} \): \( \text{Salary} \);
2. \( \text{array}[1..n] \) of \( \text{var} \ 1..w \): \( \text{Worker} \); % job \( j \) by \( \text{Worker}[j] \)
3. solve minimize \( \sum(\text{j in 1..n})(\text{Salary}[\text{Worker}[\text{j}]]); \)
4. constraint \( ... \); % qualifications, workload, etc

Example (Travelling salesperson problem)

1. \( \text{array}[1..n,1..n] \) of \( \text{float} \): \( \text{Distance} \); % instance data
2. \( \text{array}[1..n] \) of \( \text{var} \ 1..n \): \( \text{Next} \); % go from \( c \) to \( \text{Next}[c] \)
3. solve minimize \( \sum(\text{c in 1..n})(\text{Distance}[\text{c,Next}[\text{c}]]); \)
4. constraint \( \text{circuit}(\text{Next}); \)
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?

One of the many models:

- **Use one variable**, $\text{Row}[q]$, for each queen $q$ in $1..8$: queen $q$ is to be placed in column $q$.
  The **constraint** that all queens are in distinct columns is automatically **satisfied** by the choice of variables!

- **Row[$q$]** represents the row of queen $q$:
  the **domain** of each variable $\text{Row}[q]$ is $1..8$.
  Ex: $\text{Row}[3]=2$ if the queen of column 3 is in row 2.

- The remaining **constraints** to be **satisfied** are:
  - All queens are in distinct rows:
    the variables $\text{Row}[q]$ take distinct values for all $q$.
  - All queens are in distinct diagonals:
    the expressions $\text{Row}[q]+q$ take distinct values for all $q$;
    the expressions $\text{Row}[q]-q$ take distinct values for all $q$. 
An 8-Queens Model in MiniZinc

Model file 8-queens.mzn:

```plaintext
% Model of the 8-queens problem
include "globals.mzn";

int: n = 8;  % parameter
array[1..n] of var 1..n: Row;  % variables & domains

costain alldifferent( Row );
constraint alldifferent([Row[q]+q | q in 1..n]);
constraint alldifferent([Row[q]-q | q in 1..n]);
solve satisfy;  % objective
output [show(Row)];
```

The alldifferent constraint predicate requires that all 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 $q$ 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.

- **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.
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
- . . .
Constraint-Based Modelling

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

- There are several **types** for variables: integers (**int**), reals (**float**), Booleans (**bool**), strings (**string**), integer sets (**set**), and matrices thereof (**array**).


- 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
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:

```
mzn-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 (Constructive 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, CP, LCG, and MIP solvers.

Definition (Perturbative Search)

Start from a candidate solution and iteratively modify it. It is the basic idea behind LS and GA solvers.
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, GA, and ACO are rather methodologies.
- Some technologies have standardised modelling languages across all solvers: SAT, SMT, 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:

– Need to learn one or more modelling languages?
– Need to understand the used solving technologies in order to get the most out of them?
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 one or more modelling languages?
– Need to understand the used solving technologies in order to get the most out of them?
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 one or more 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
MiniZinc is a declarative language (not a solver) for the constraint-based modelling of constraint problems:

- Mainly at U of Melbourne and Monash U, Australia.
- Introduced in 2007; version 2.0 in 2014.
- Nice integrated development environment (IDE).
- Annual MiniZinc Challenge for solvers, since 2008.
- There are also courses at Coursera.
MiniZinc Features

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

- SAT = Boolean satisfiability: Plingeling via PicatSAT, ...
- SMT = SAT modulo theories: Yices, ... via fzn2smt
- MIP = mixed integer programming: Cbc, IBM ILOG CPLEX Optimizer, G12.MIP, Gurobi Optimizer, ...
- CP = constraint programming: Choco, G12.FD, Gecode, JaCoP, Mistral, SICStus Prolog, ...
- CBLS = constraint-based LS (local search): OscaR.cbls via fzn-oscar-cbls, Yuck, ...
- LCG = lazy clause generation = CP + SAT: Chuffed, G12.lazyFD, Google OR-Tools, Opturion CPX, ...
- Other hybrid technos: iZplus, MiniSAT(ID), SCIP, ...
- Portfolios of solvers: sunny-cp, ...
Solvers with MiniZinc Backends

- **SAT** = Boolean satisfiability: Plingeling via PicatSAT, ...
- **SMT** = SAT modulo theories: Yices, ... via fzn2smt
- **MIP** = mixed integer programming: Cbc, IBM ILOG CPLEX Optimizer, G12.MIP, Gurobi Optimizer, ...
- **CP** = constraint programming: Choco, G12.FD, Gecode, JaCoP, Mistral, SICStus Prolog, ...
- **CBLS** = constraint-based LS (local search): OscaR.cbls via fzn-oscar-cbls, Yuck, ...
- **LCG** = lazy clause generation = CP + SAT: Chuffed, G12.lazyFD, Google OR-Tools, Opturion CPX, ...
- **Other hybrid technos**: iZplus, MiniSAT(ID), SCIP, ...
- **Portfolios of solvers**: sunny-cp, ...

Backends installed on IT dept’s ThinLinc hardware are red.
## MiniZinc Challenge 2015: Some Winners

<table>
<thead>
<tr>
<th>Problem &amp; Model</th>
<th>Backend</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>MZN / 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)
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 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 \( \text{var 4..6: x} \) is

\[
\begin{align*}
\text{array[4..7] of var bool: B; } & \quad \% \text{ B[i] denotes truth of x \( \geq i \) } \\
\text{constraint B[4]; } & \quad \% \text{ lower bound on x } \\
\text{constraint not B[7]; } & \quad \% \text{ upper bound on x } \\
\text{constraint B[4] \lor not B[5]; } & \quad \% \text{ consistency } \\
\text{constraint B[5] \lor not B[6]; } & \quad \% \text{ consistency } \\
\text{constraint B[6] \lor not B[7]; } & \quad \% \text{ consistency }
\end{align*}
\]

For an integer variable with \( n \) domain values, there are \( n + 1 \) Boolean variables and \( n \) clauses, all over 2 variables.
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**

```plaintext
alldifferent([x, y, z]) gives x!=y/\y!=z/\z!=x.
```

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

**Examples (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 5: Solving Technologies.

+ For most users, the time to achieve a particular solving speed or solution quality (or both) is drastically reduced by using a model-once-solve-everywhere toolchain, compared to programming from first principles, without knowing (deeply) how the solvers work.
How to Solve a Combinatorial 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 Combinatorial 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 Combinatorial 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
Objective, from the Catalogue (1DL448)

Learn the use of tools to solve hard combinatorial optimisation problems by first modelling them in a solving-technology-independent constraint-based modelling language and then using an off-the-shelf constraint solver, as opposed to designing an (approximation) algorithm from first principles.
Learning Outcomes, from the Catalogue

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

- model a combinatorial problem using a solving-technology-independent modelling language;
- discuss various models of a combinatorial problem expressed in a constraint-based modelling language;
- describe and compare solving technologies that can be used by the backends to a modelling language, including constraint programming (CP), local search (LS), Boolean satisfiability (modulo theories) (SAT and SMT), and mixed integer programming (MIP);
- decide which solving technologies to try when facing a new combinatorial problem, and motivate this decision;
- design and evaluate different models of a combinatorial problem for various solving technologies.
Topics

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

- Lectures, given in English

- No textbook: slides, MiniZinc documentation, Coursera

- 2 lab sessions for familiarisation with MiniZinc toolchain

- 3 teacher-chosen assignments with help sessions, to be done in student-chosen solo / duo team (3 credits)

- 1 optional student-chosen project, to be done in student-chosen solo / duo team (2 credits)
Project (2 credits)

- Model and solve a combinatorial problem that you are interested in, ideally for your PhD research.

- Ask us, or see for example http://CSPlib.org for problems and instance data.

- Hand in a report in the style of those for the assignments, whenever you are ready.

- We may schedule presentation session(s) where everyone presents their project to the class.
We use the MiniZinc language and toolchain for this course:

- All backends we recommend are installed on the Linux ThinLinc computers of the IT department.

- You can also install it all by yourself directly on your own machine; this can be tedious.


The commercial solvers are under free academic licenses: you may never use them for non-academic purposes.