Topic 1: Introduction[1]
(Version of 26th August 2019)

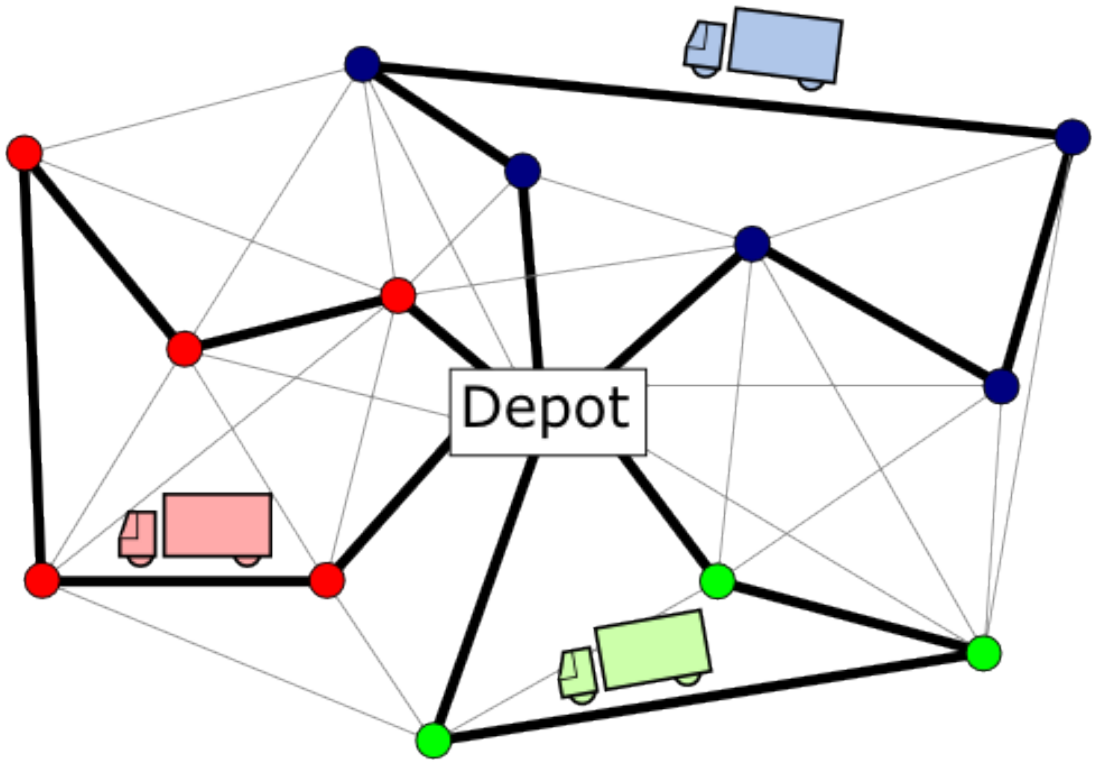
Pierre Flener and Jean-Noël Monette

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL441:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation

[1] Based partly on material by Guido Tack

Optimisation



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

UPPSALA UNIVERSITET

MiniZinc Challenge 2015: Some Winners

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

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Outline

Example (Agricultural experiment design)

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

Constraints to be **satisfied**:

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

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

Example (Agricultural experiment design)

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

Constraints to be **satisfied**:

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

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

Example (Doctor rostering)

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

Constraints to be **satisfied**:

1. #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)

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

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**:

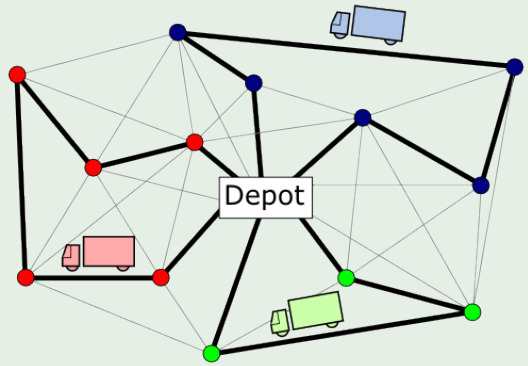
- Cost: ...

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**:

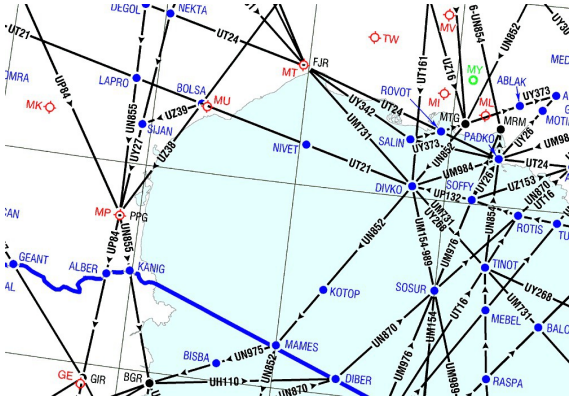
- Cost: the total fuel consumption and driver salary.

Example (Travelling salesperson: optimisation TSP)

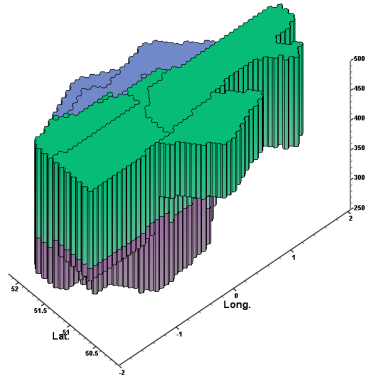
Given a map and cities, **find** a **shortest** route visiting each city once and returning to the starting city.

Applications in Air Traffic Management

Demand vs capacity



Airspace sectorisation



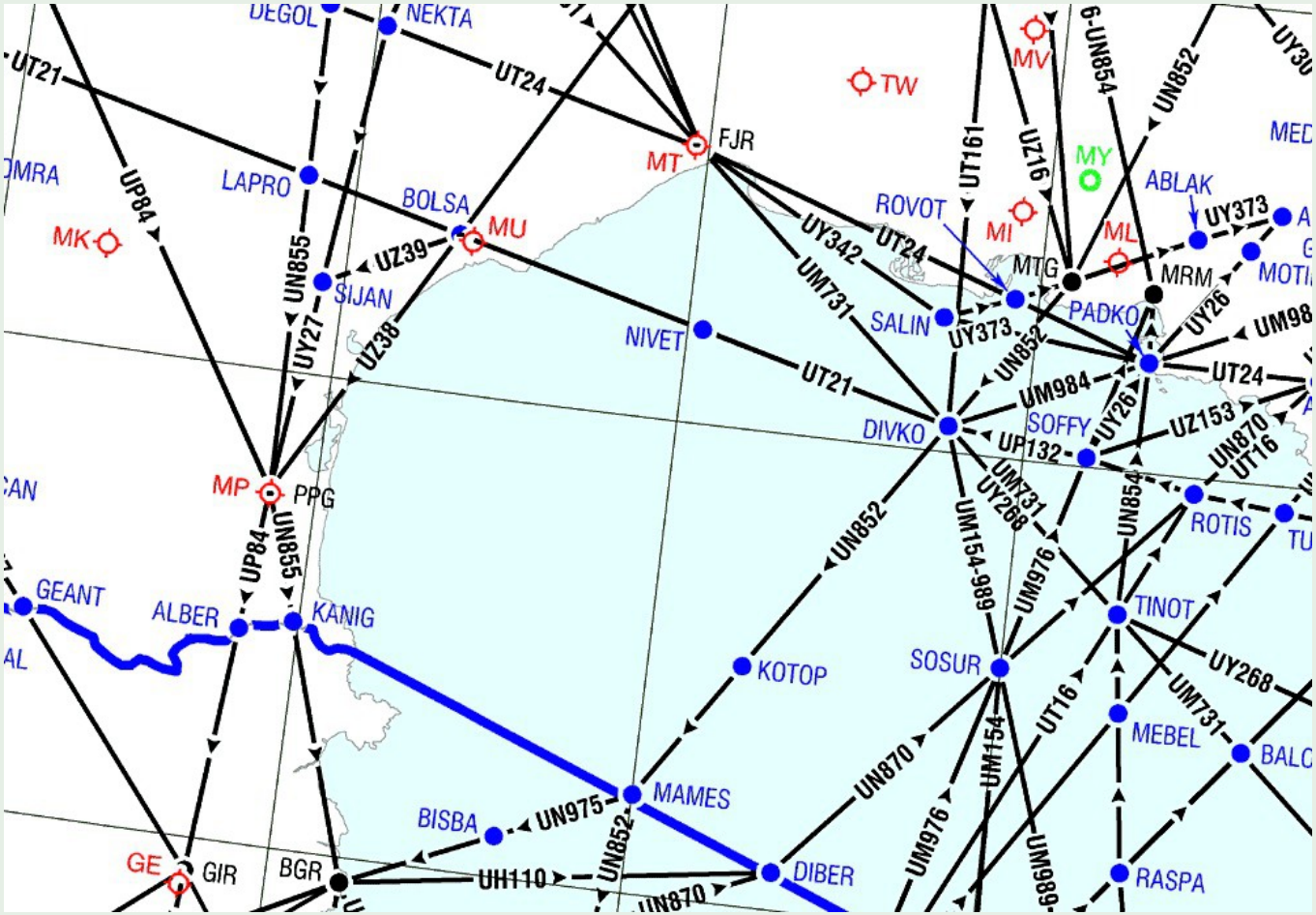
Contingency planning

Flow	Time Span	Hourly Rate
From: Arlanda To: west, south	00:00 – 09:00 09:00 – 18:00 18:00 – 24:00	3 5 2
From: Arlanda To: east, north	00:00 – 12:00 12:00 – 24:00	4 3
...	...	...

Workload balancing

Example (Air-traffic demand-capacity balancing)

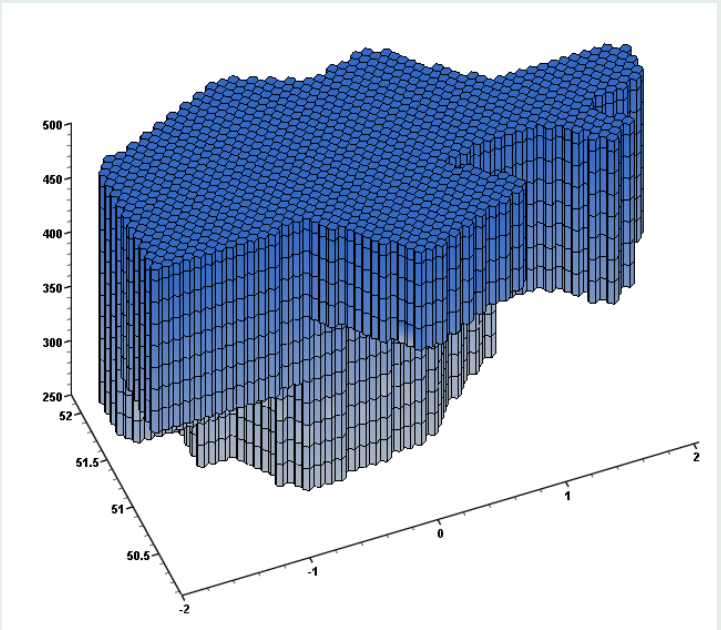
Reroute flights, in height and speed, so as to balance the workload of air traffic controllers in a multi-sector airspace:

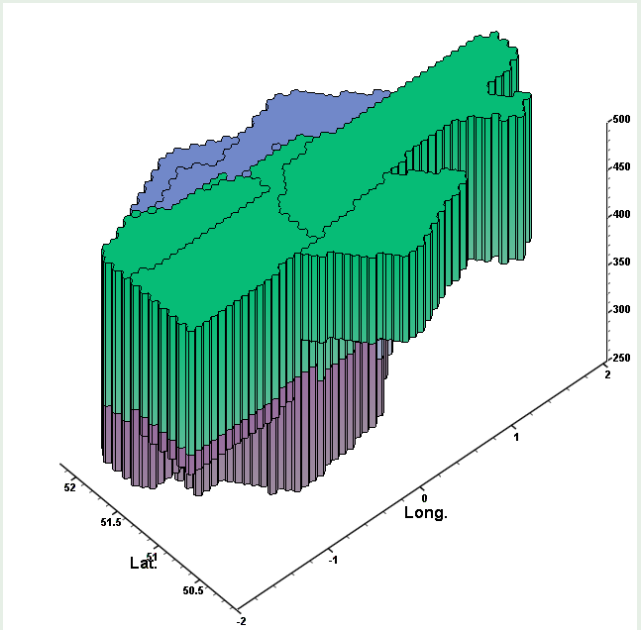


Example (Airspace sectorisation)

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

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

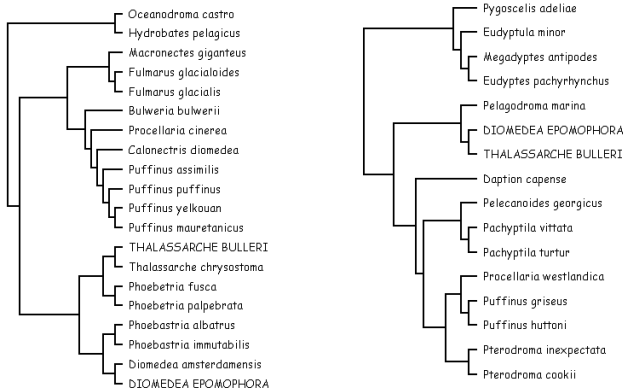




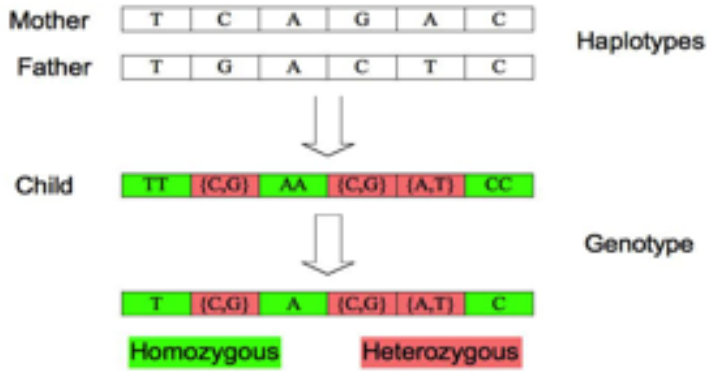
There are s^c possible colourings, but very few optimally satisfy the constraints: is intelligent search necessary?

Applications in Biology and Medicine

Phylogenetic supertree



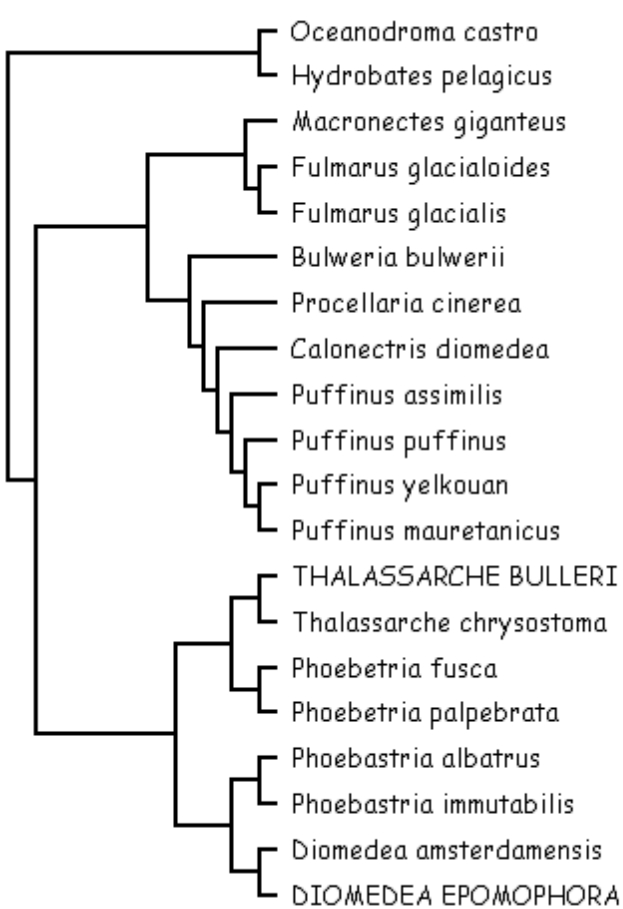
Haplotype inference

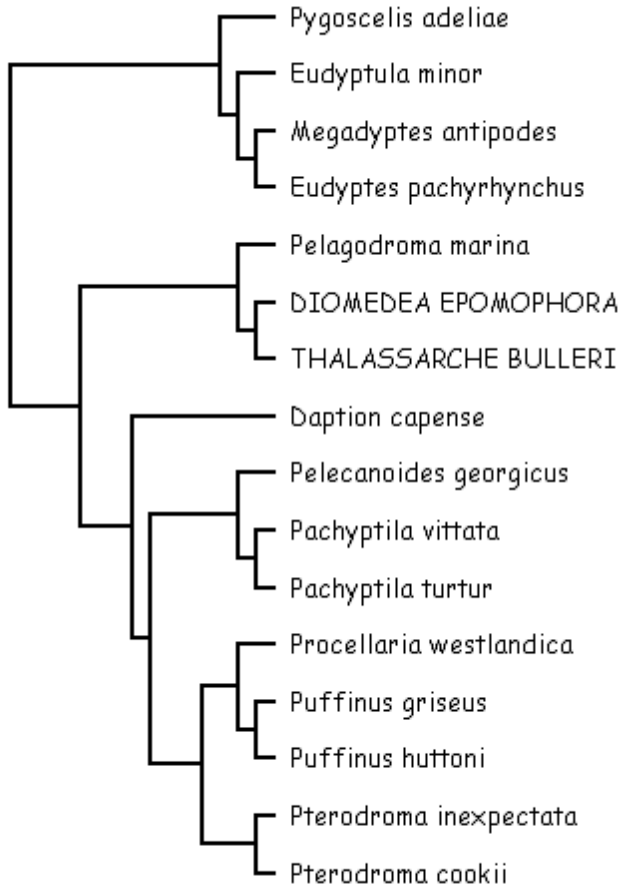


Medical image analysis

Doctor rostering

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





Example (Haplotype inference by pure parsimony)

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

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

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

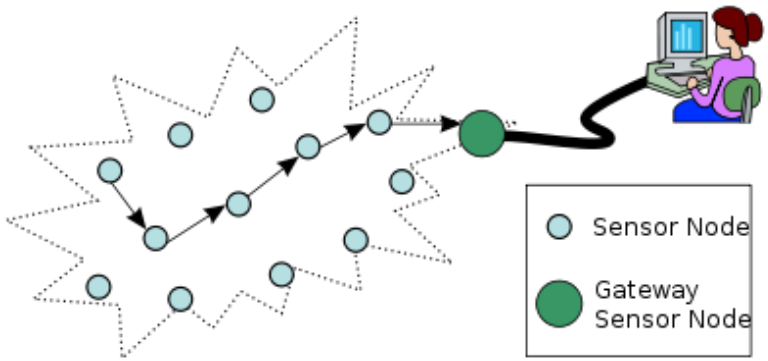
...					
A	C	T	C	T	C
...					
A	G	T	C	A	C
...					
T	G	T	G	A	C
...					

so that each given genotype conflates 2 found haplotypes.

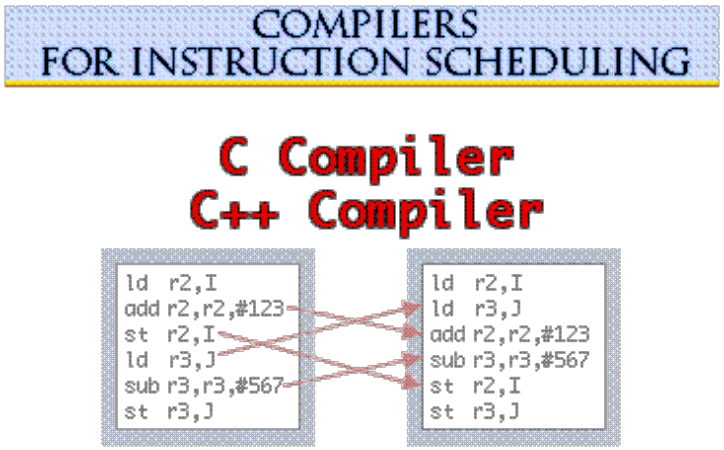
Applications in Programming and Testing

Robot-task sequencing

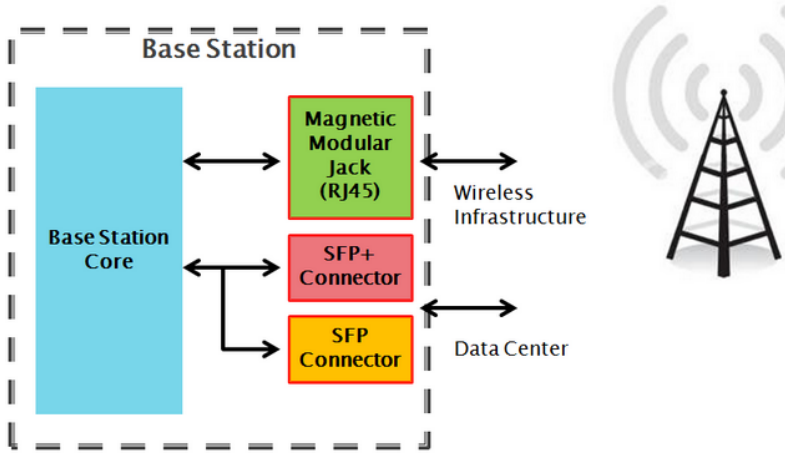
Sensor-net configuration



Compiler design



Base-station testing



Other Application Areas

School timetabling

	Monday	Tuesday	Wednesday	Thursday	Friday
9.00	MT2202 Ordinary Differential Equations FTh1		LABC52072 Computer Graphics (G) Dual	MT2282 Numerical Analysis II Williamson, G03	
10.00	XMT2202 Ordinary Differential Equations NG15 / Roscoe, 2.3		LABC52072 Computer Graphics (G) Dual	XMT2202 Ordinary Differential Equations Simon Engineering, Basement Theatre 3A XMT2282 Numerical Analysis II LG20	XMT2202 Ordinary Differential Equations NG15
11.00	CS2012 Algorithms and Data Structures 1.1		XMT2212 Further Linear Algebra 1.5		MT2202 Ordinary Differential Equations Stopford, Theatre 1
12.00	MT2212 Further Linear Algebra Roscoe, Theatre A	MT2282 Numerical Analysis II Williamson, G03	CS2072 Computer Graphics 1.1		MT2212 Further Linear Algebra Stopford, Theatre 1
1.00			PASS Peer Assisted Study I407 / LF15 / LF17 / I406		XMT2212 Further Linear Algebra Simon Engineering, Basement Theatre 4A
2.00	CS2072 Computer Graphics 1.1			XMT2212 Further Linear Algebra NG17	
3.00		CSTUT Tutorial			
4.00		CS2012 Algorithms and Data Structures 1.1			

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.

$P \stackrel{?}{=} NP$ (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$. 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 ($P \neq NP$)?

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 ℓ of edges is NP-complete. Hence finding a longest path seems hard: increase ℓ 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.

UPPSALA UNIVERSITET

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 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:

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

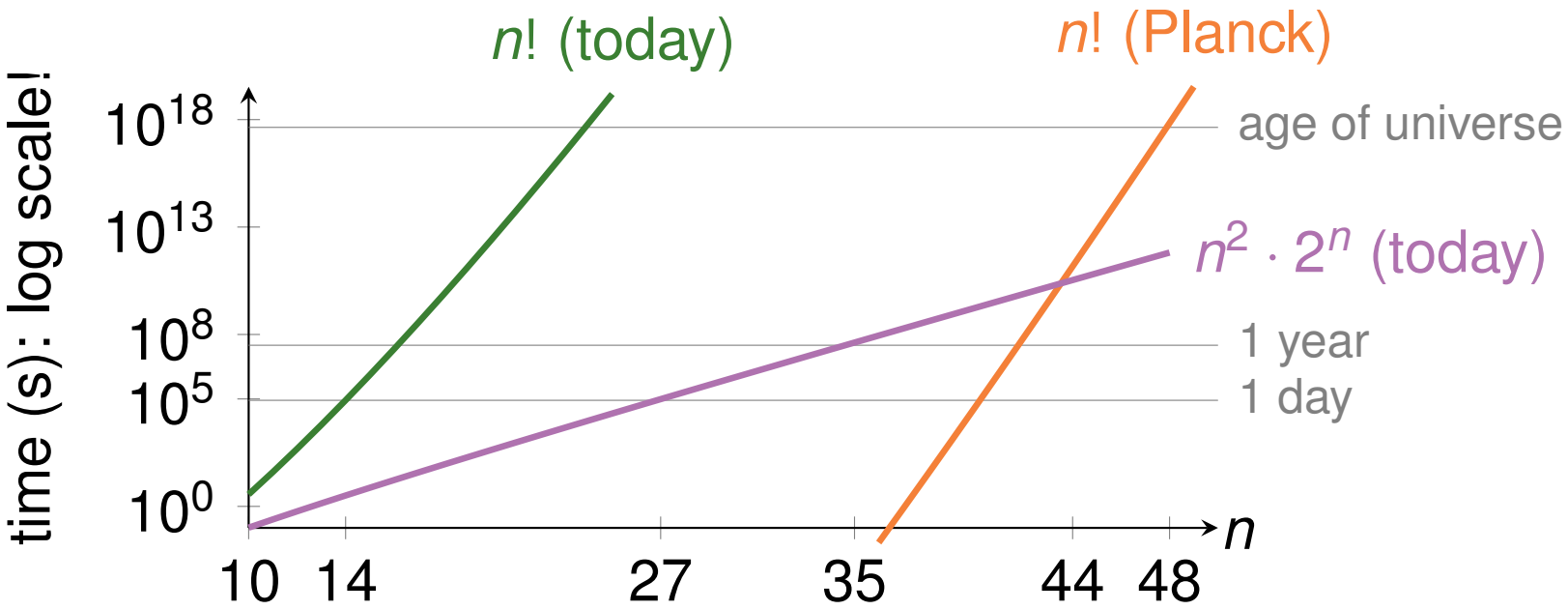
- Planck time is shortest useful interval: $\approx 5.4 \cdot 10^{-44}$ s; a Planck computer would evaluate $1.8 \cdot 10^{43}$ routes / s:

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

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

UPPSALA UNIVERSITET

Outline

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 & 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 a model as input:

- Boolean satisfiability (SAT)
- SAT modulo theories (SMT)
- (Mixed) integer linear programming (IP and MIP)
- Constraint programming (CP) ☞ part 2 of 1DL441
- ...
- Hybrid technologies (LCG $=$ CP $+$ SAT, ...)

Methodologies, *usually without* modelling and solvers:

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

UPPSALA UNIVERSITET

Outline

What vs How

Example

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

A **formal specification** is:

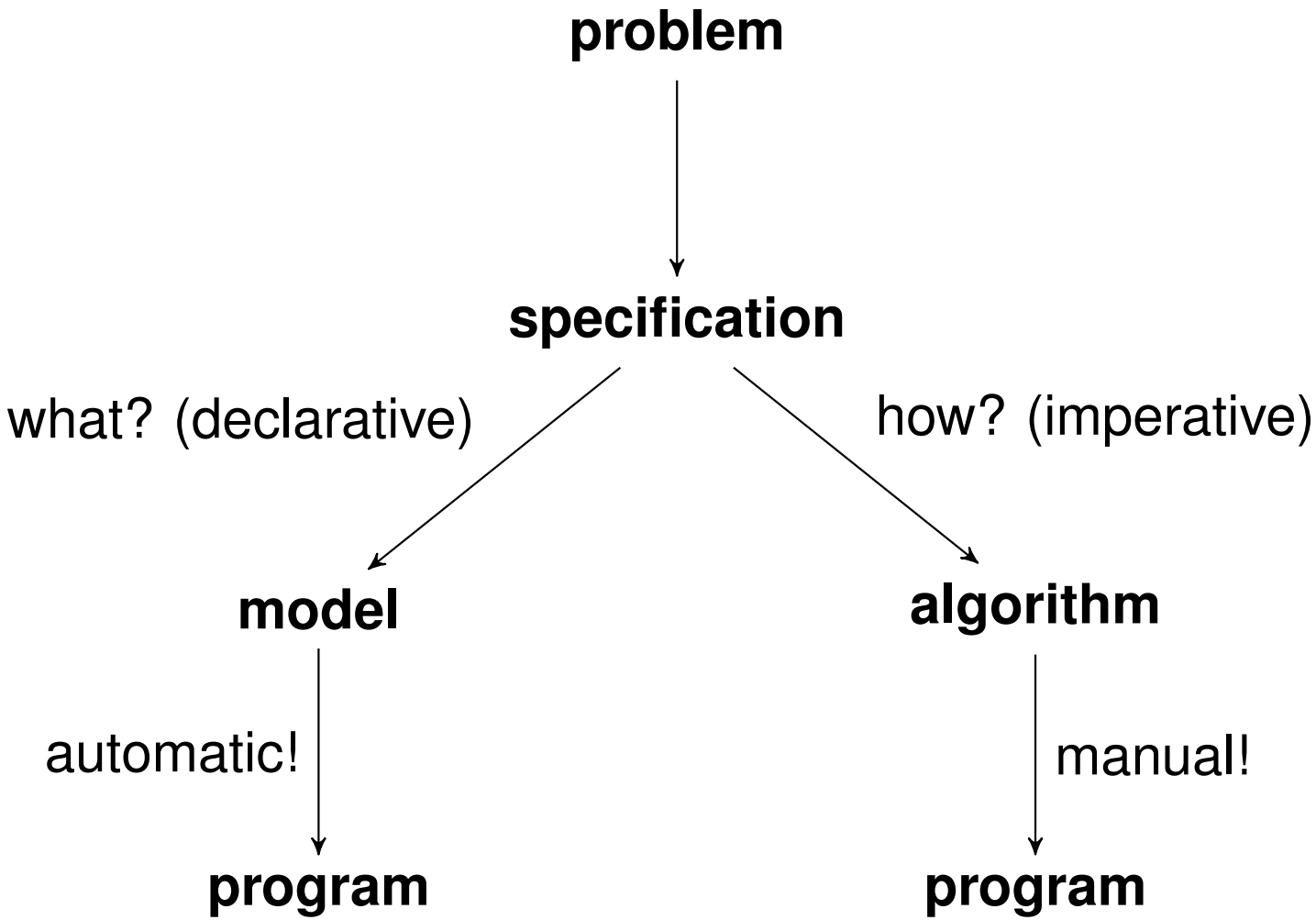
$$\text{sort}(A, S) \equiv \text{permutation}(A, S) \wedge \text{increasing}(S)$$

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

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

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

Modelling vs Programming



Example (Sudoku)

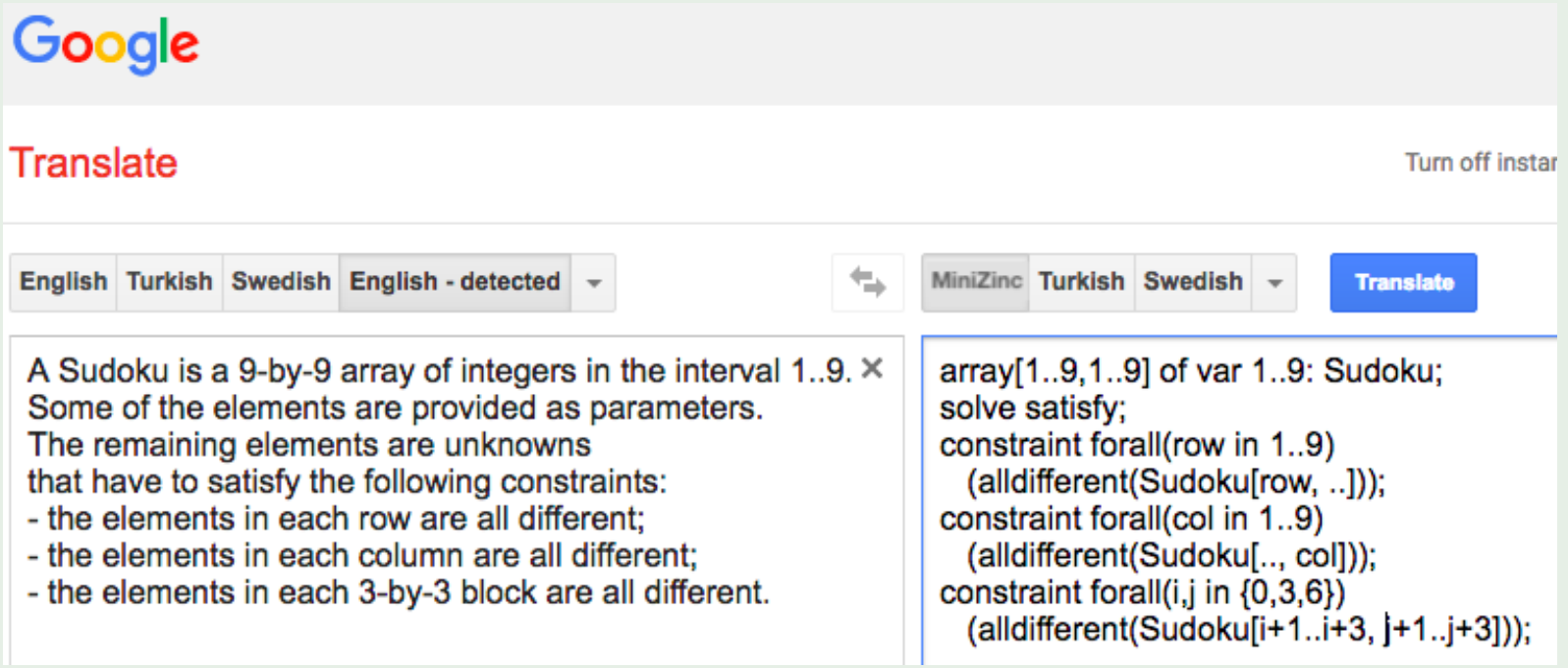
8								
		3	6					
	7			9		2		
	5				7			
				4	5	7		
			1				3	
		1					6	8
		8	5				1	
	9					4		

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

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:

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)



Example (Sudoku)

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

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

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

Example (Agricultural experiment design, AED)

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

Constraints to be **satisfied**:

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

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

General term: balanced incomplete block design (BIBD).

Example (Agricultural experiment design, AED)

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

Constraints to be **satisfied**:

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

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

General term: balanced incomplete block design (BIBD).

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

Example (BIBD *integer* model: ✓ $\rightsquigarrow$ 1 and – $\rightsquigarrow$ 0)

```
-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)

```
-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: ✓ $\rightsquigarrow$ 1 and – $\rightsquigarrow$ 0)

```
-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 = count(BIBD[..,b], 1));
 2 constraint forall(v in Varieties)
     (sampleSize = count(BIBD[v,..], 1));
 3 constraint forall(v, w in Varieties where v < w)
     (balance = count([BIBD[v,b]*BIBD[w,b] | b in Blocks], 1));
```

Example (Instance data for our AED)

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

Using the `count` abstraction over linear expressions:

Example (BIBD *integer* model: ✓ $\rightsquigarrow$ 1 and – $\rightsquigarrow$ 0)

```
-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 = count(BIBD[..,b], 1));
 2 constraint forall(v in Varieties)
     (sampleSize = count(BIBD[v,..], 1));
 3 constraint forall(v, w in Varieties where v < w)
     (balance = count([BIBD[v,b]+BIBD[w,b] | b in Blocks], 2));
```

Example (Instance data for our AED)

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

Reconsider the model fragment:

```
3 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] * 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] * 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)

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

Constraints to be **satisfied**:

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

Example (BIBD *set* model: a block *set* per variety)

```
-3 enum Varieties; enum Blocks;
-2 int: blockSize; int: sampleSize; int: balance;
-1 array[Varieties] of var set of Blocks: BIBD;
 0 solve satisfy;
 1 constraint forall(b in Blocks)
     (blockSize = sum(v in Varieties)(bool2int(b in BIBD[v])));
 2 constraint forall(v in Varieties)
     (sampleSize = card(BIBD[v]));
 3 constraint forall(v, w in Varieties where v < w)
     (balance = card(BIBD[v] intersect BIBD[w]));
```

Example (Instance data for our AED)

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

Example (Doctor rostering)

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

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)

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

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)

```
-3 enum Days; enum SurgeryDays; enum Doctors;
-2 enum ShiftTypes = {app, 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 SurgeryDays)
     (count(Roster[..,d],oper) <= 2);
 3 constraint count(Roster,oper) >= 7;
 4 constraint count(Roster,app) >= 4;
 5 constraint forall(d in Doctors)
     (regular(Roster[d,..], "((oper none)|app|call|none)*"));
 6 ...  % other constraints
```

Example (Instance data for our hospital unit)

```
Days = {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
SurgeryDays = {Mon, Tue, Wed, Thu, Fri};
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)

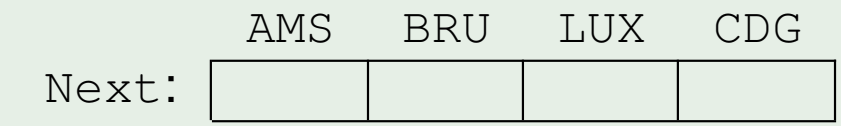
Given jobs `Jobs` and the salaries of work applicants `Apps`,
find a work applicant for each job
such that some constraints (on the qualifications of the work applicants for the jobs, on workload distribution, etc) are satisfied and the total salary cost is minimal:

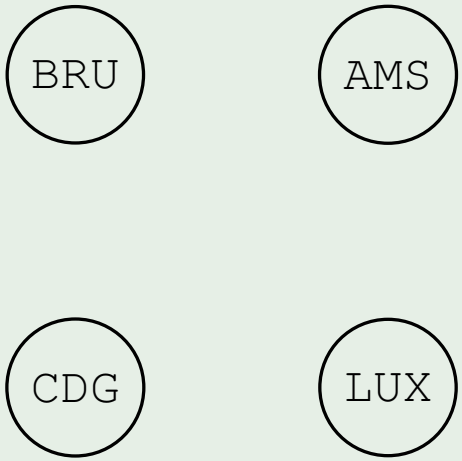
```
1 array[Apps] of int: Salary;
2 array[Jobs] of var Apps: Worker; % job j by Worker[j]
3 solve minimize sum(j in Jobs)(Salary[Worker[j]]);
4 constraint ...;          % qualifications, workload, etc
```

Using variables as indices within arrays: black magic?!

Example (Vehicle routing: backbone model)

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





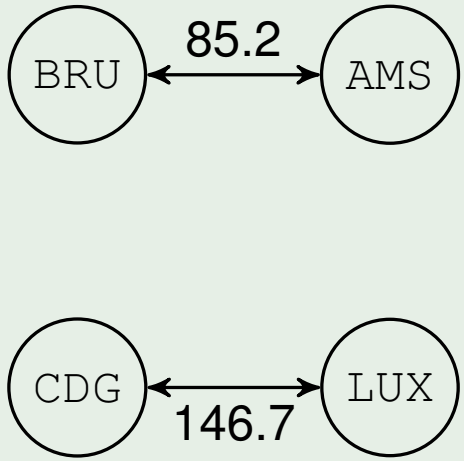
Using variables as indices within arrays: black magic?!

Example (Vehicle routing: backbone model)

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

	AMS	BRU	LUX	CDG
Next:	BRU	AMS	CDG	LUX

So `alldifferent(Next)` is too weak!



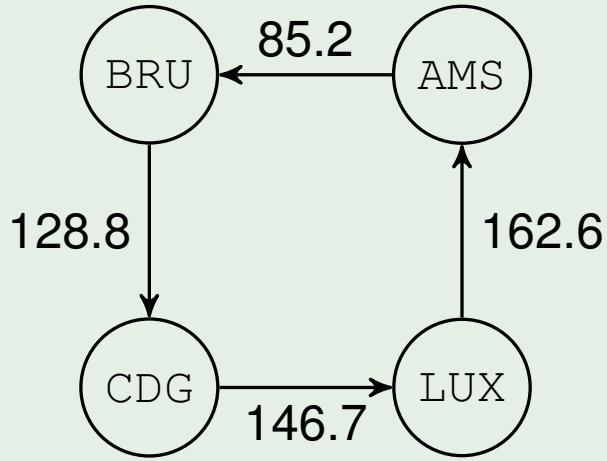
Using variables as indices within arrays: black magic?!

Example (Vehicle routing: backbone model)

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

	AMS	BRU	LUX	CDG
Next:	BRU	CDG	AMS	LUX

Let us use `circuit(Next)` instead:

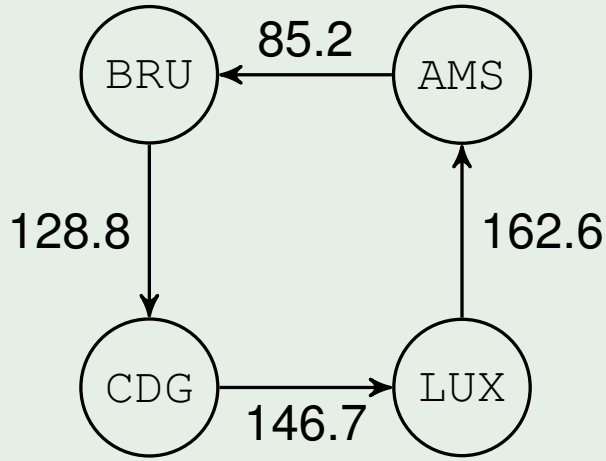


Using variables as indices within arrays: black magic?!

Example (Vehicle routing: backbone model)

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

	AMS	BRU	LUX	CDG
Next:	BRU	CDG	AMS	LUX

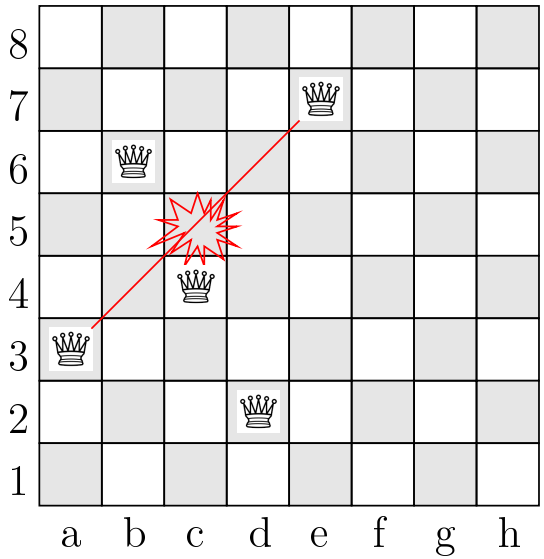


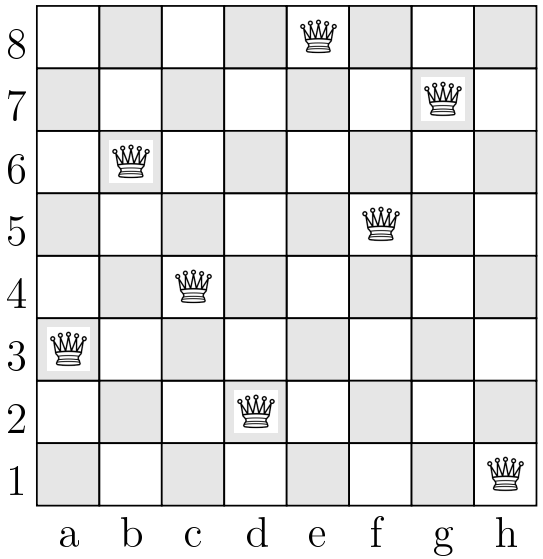
Let us use `circuit(Next)` instead:

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

Toy Example: 8-Queens

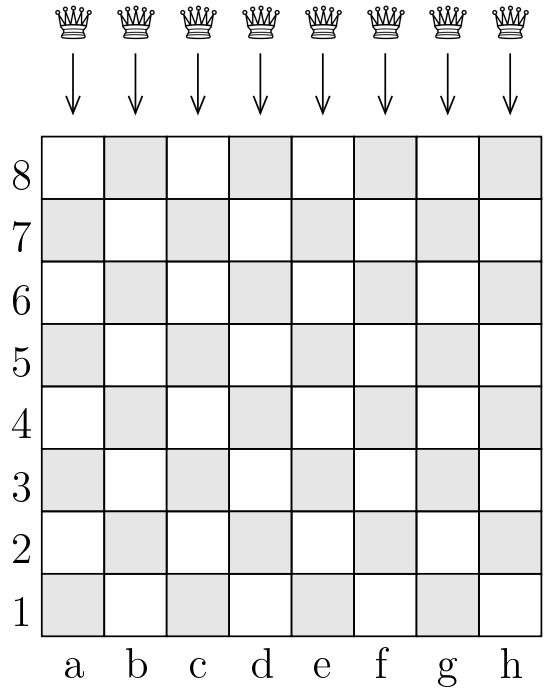
Can one place 8 queens onto an 8×8 chessboard so that all queens are in distinct rows, columns, and diagonals?





An 8-Queens Model

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



- Let **variable** `Row[q]`, of **domain** `1..8`, **represent** the row of the queen in column `q`, for `q` in `a..h`, renamed into `1..8`.
  Example: `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 `Row[q]` take distinct values for all `q`.
  - All queens are in distinct diagonals:
    the expressions `Row[q]+q` take distinct values for all `q`;
    the expressions `Row[q]-q` take distinct values for all `q`.

An 8-Queens Model in MiniZinc

Consider the following model in file `8-queens.mzn`:

```
1  % Model of the 8-queens problem
2  include "globals.mzn";
3  % parameter:
4  int: n = 8; % n denotes the given number of queens
5  % Row[q] denotes the row of the queen in column q:
6  array[1..n] of var 1..n: Row; % variables and domains
7  % constraints:
8  constraint alldifferent( Row                    );
9  constraint alldifferent([Row[q]+q | q in 1..n]);
10 constraint alldifferent([Row[q]-q | q in 1..n]);
11 % objective:
12 solve satisfy; % solve to satisfaction
13 % pretty-printing of solutions:
14 output [show(Row)];
```

The `alldifferent` 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 $\mathrm{dom}(x)$, 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.

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
  - ...

Constraint-Based Modelling

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 multidimensionsal matrices (`array`).
- There is a nice vocabulary of **predicates** (`<`, `<=`, `=`, `!=`, `>=`, `>`, `alldifferent`, `circuit`, `regular`, ...), **functions** (`+`, `-`, `*`, `card`, `count`, `inter`, `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.

UPPSALA UNIVERSITET

Outline

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.

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, 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 several modelling languages? No!
– Need to understand the used solving technologies in order to get the most out of them? Yes, but . . . !

UPPSALA UNIVERSITET

Outline

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

UPPSALA UNIVERSITET

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

Solvers with MiniZinc Backends

- SAT = Boolean satisfiability: Plingeling via PicatSAT, ...
- SMT = SAT modulo theories: Yices, . . . via fzn2smt
- 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 = CP + SAT: Chuffed, 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, 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 = CP + SAT: Chuffed, Google OR-Tools, Opturion CPX, ...
- Other hybrid technos: iZplus, MiniSAT(ID), SCIP, ...
- Portfolios of solvers: sunny-cp, ...

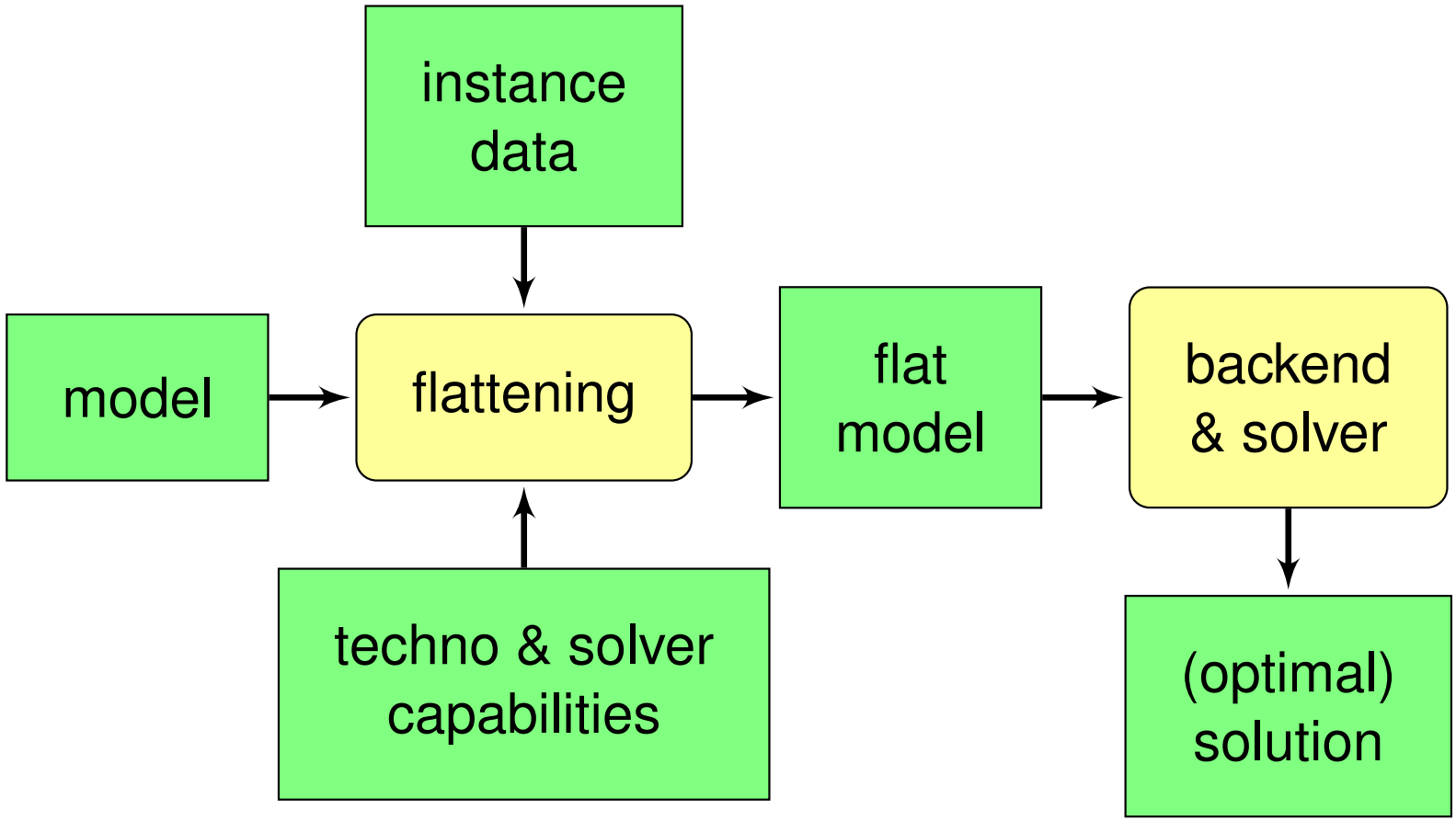
Backends installed on IT dept's ThinLinc hardware are red. The commercial Gurobi Optimizer is under a free academic license: you may not use it for non-academic purposes.

MiniZinc Challenge 2015: Some Winners

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

(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 `var 4..6: x` is

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

`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.

Example (IP and MIP)

A compact linearisation of `x != y` is

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

One cannot naturally model graph colouring in IP, but the problem has integer 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 Combinatorial Problem?

1. Model the problem

2. Solve the problem

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!

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Outline

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.

UPPSALA UNIVERSITET

Learning Outcomes of Part 1 (1DL451)

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 and discuss topics related to the course content, orally and in writing, with a skill appropriate for the level of education.

Organisation and Time Budget of Part 1

Period 1: September to early November, budget = 133.3 h:

- 1 student-chosen project, to be done in student-chosen duo team: budget = 42 hours / student (2 credits)
- 12 lectures, including a ***mandatory*** guest lecture, plus 3 ***mandatory*** project presentation sessions: budget = 22.5 hours
- 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 23 hours / assignment / student (3 credits)
- Prerequisites: basic algebra, combinatorics, logic, graph theory, set theory, and search algorithms

Lecture Topics of Part 1 (course 1DL451)

- 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 and SMT)
- (Topic 11: Modelling for MIP)

Project (2 credits) in Part 1 (course 1DL451)

Topic:

- Model and solve a combinatorial problem that you are interested in, say for research, a course, a hobby, ...
- Ask us, or see sites like Google Hash Code or CSPlib for problems (with no published MiniZinc or OPL models) and third-party instance data.

Students who took a course on CP are encouraged to study in advance Topic 8: Inference & Search in CP & LCG.

Deadlines:

- Wed 18 Sep at 15:00: upload project proposal
- Wed 25 Sep at 17:00: secure our project approval
- Mon 21 Oct – Tue 22 Oct: present
- Fri 01 Nov at 13:00: upload final report; score $p \in 0..10$

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

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!
- $D_i - 9$: help session a: attendance recommended
- $D_i - 7$: help session b: attendance recommended
- $D_i - 2$: help session c: attendance recommended
- $D_i \pm 0$: submission, by 13:00 Swedish time on a Friday
- $D_i + 4$ by 16:00: initial score $a_i \in 0..5$ points
- $D_i + 5$: teamwise oral grading session if $a_i \in \{1, 2\}$: possibility of earning 1 extra point for final score; otherwise final score $=$ initial score
- $D_i + 5 = D_{i+1} - 9$: 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:** $a = a_2 + a_3 \geq 50\% \cdot (5 + 5) = 5$
  ☞ the formulae for the modelling assignment grade and project grade in 3..5 are at the course homepage
- **Worth going full-blast:** An assignment sum $a \in 5..10$ is combined with a project score $p \in 5..10$ to determine the overall grade in 3..5 for 1DL451 or Part 1 of 1DL441 according to a formula at the course homepage

Students who took a course on CP are encouraged to study in advance Topic 8: Inference & Search in CP & LCG.

Assignment and Project Rules

Register teams by Sun 8 Sep at 23:59 at Student Portal:

- **Duo teams:** Two consenting partners sign up at portal
- **Solo teams:** Apply to head teacher, who rarely agrees
- **Random partner?** Assent to TA, else you're bounced

Other considerations:

- **Why (not) like this? Why no email reply?** See FAQ
- **Partner swapping:** Allowed, but to be declared to TA
- **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

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

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 and discuss topics related to the course content, orally and in writing, with a skill appropriate for the level of education.

Organisation and Time Budget of Part 2

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

- 12 lectures, including a ***mandatory*** guest lecture: budget = 19.5 hours
- No textbook: slides and Gecode documentation
- 1 warm-up session for learning the Gecode 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 38 hours / assignment / student (5 credits)
- Prerequisites: C++; basic algebra, combinatorics, logic, graph theory, set theory, and search algorithms

UPPSALA UNIVERSITET

Lecture Topics of Part 2

- Topic 12: CP and Gecode
- Topic 13: Consistency
- Topic 14: Propagation
- Topic 15: Search
- Topic 16: Propagators
- Topic 17: Constraint-Based Local Search
- Topic 18: Conclusion

3 Assignment Cycles of 2–3 Weeks in Part 2

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: attendance recommended
- $D_i - 3$: help session b: attendance recommended
- $D_i - 1$: help session c: attendance recommended
- $D_i \pm 0$: submission, by 13:00 Swedish time on a Friday
- $D_i + 6$ by 16:00: initial score $a_i \in 0..5$ points
- $D_i + 7$: teamwise oral grading session if $a_i \in \{1, 2\}$: possibility of earning 1 extra point for final score; otherwise final score $=$ initial score
- $D_i + 7 = D_{i+1} - 7$: solution session & help session a

Assignments (5 c) in Part 2 & Overall Grade

Let $a_i \in 0..5$ be final score on Assignment i, with $i \in 4..6$:

- **20% threshold:** $\forall i \in 4..6 : a_i \geq 20\% \cdot 5 = 1$
  No catastrophic failure on individual assignments
- **50% threshold:** $a_4 + a_5 + a_6 \geq \lceil 50\% \cdot (5+5+5) \rceil = 8$
  ☞ the formula for the programming assignment grade in 3..5 is at the course homepage
- **Worth going full-blast:** An overall grade $m \in 3..5$ for Part 1 is combined with a programming assignment grade $c \in 3..5$ for Part 2 in order to determine the overall course grade in 3..5 for 1DL441 according to a formula at the course homepage

Assignment Rules

Register teams by Sun 10 Nov at 23:59 at Student Portal:

- **Duo teams:** Two consenting partners sign up at portal
- **Solo teams:** Apply to head teacher, who rarely agrees
- **Random partner?** Assent to TA, else you're bounced

Other considerations:

- **Why (not) like this? Why no email reply?** See FAQ
- **Partner swapping:** Allowed, but to be declared to TA
- **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

UPPSALA UNIVERSITET

Outline

How To Communicate by Email?

To email the assistants or the head teacher:

- For 1DL451 and part 1 of 1DL441, write to the head teacher, or the assistant, or both, as per below.
- For part 2 of 1DL441, you must use the email list COCP.help@gmail.com, which is monitored by all the assistants only during period 2.
- If you have a question about the **lecture material** or **course organisation**, then contact the head teacher.
- If you have a question about the **assignments** or **infrastructure**, then contact an assistant at a help session for an immediate answer. Emailed short questions triggering short reply times will be answered as soon as possible during normal work hours, otherwise you may be pointed to the next help session.

UPPSALA UNIVERSITET

What Has Changed Since Last Time?

Changes wanted by the head teacher:

- Shorter reports (but equal model-tuning expectation)
- No draft report and no opposition for M4CO project

Changes triggered by course evaluations:

- Change log on pages & documents of the website
- Assignments 1 to 3: now 3 help sessions (instead of 2)
- Assignments 2+5: now 3 weeks allocated (instead of 2)
- M4CO project: now 2 help sessions (instead of 1)
- Assignment 6: now has a question on local search