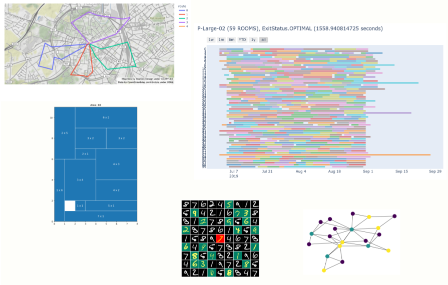
L01: Intro to Model+Solve and Combinatorial Optimisation



Prof. Tias Guns and Dr Dimos Tsouros

KU LEUVEN

Based on slides from Pierre Flener, Uppsala University + slides from Guido Tack & Chris Beck.

Example (From: Steven S. Skiena's The Algorithm Design Manual)

Imagine you are a highly-in- demand actor, who has been presented with offers to star in n different movie projects under development. Each offer comes specified with the first and last day of filming. To take the job, you must commit to being available throughout this entire period. Thus, you cannot simultaneously accept two jobs whose intervals overlap.

For an artist such as yourself, the criteria for job acceptance is clear: you want to make as much money as possible. Because each of these films pays the same fee per film, this implies you seek the largest possible set of jobs (intervals) such that no two of them conflict with each other.

Title	Start	End
Tarjan of the Jungle	4	13
The Four Volume Problem	17	27
The President's Algorist	1	10
Steiner's Tree	12	18
Process Terminated	23	30
Halting State	9	16
Programming Challenges	19	25
Discrete Mathematics	2	7
Calculated Bets	26	31

How would you solve this problem?

Example (From: Steven S. Skiena's The Algorithm Design Manual)

Imagine you are a highly-in- demand actor, who has been presented with offers to star in n different movie projects under development. Each offer comes specified with the first and last day of filming. To take the job, you must commit to being available throughout this entire period. Thus, you cannot simultaneously accept two jobs whose intervals overlap.

For an artist such as yourself, the criteria for job acceptance is clear: you want to make as much money as possible. Because each of these films pays the same fee per film, this implies you seek the largest possible set of jobs (intervals) such that no two of them conflict with each other.

Title	**Start**	**End**
...	...	...

How would you solve this problem?

1. Trial and error on paper
2. Write a custom search algorithm
3. Reuse an existing, generic problem solving paradigm

Example (From: Steven S. Skiena's The Algorithm Design Manual)

Imagine you are a highly-in- demand actor, who has been presented with offers to star in n different movie projects under development. Each offer comes specified with the first and last day of filming. To take the job, you must commit to being available throughout this entire period. Thus, you cannot simultaneously accept two jobs whose intervals overlap.

For an artist such as yourself, the criteria for job acceptance is clear: you want to make as much money as possible. Because each of these films pays the same fee per film, this implies you seek the largest possible set of jobs (intervals) such that no two of them conflict with each other.

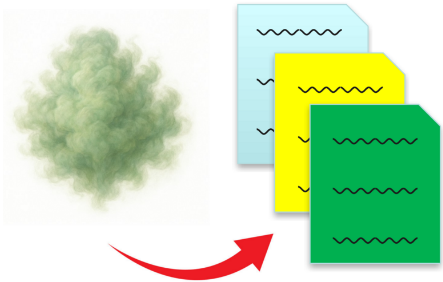
Title	Start	End
...	...	...

How would you solve this problem?

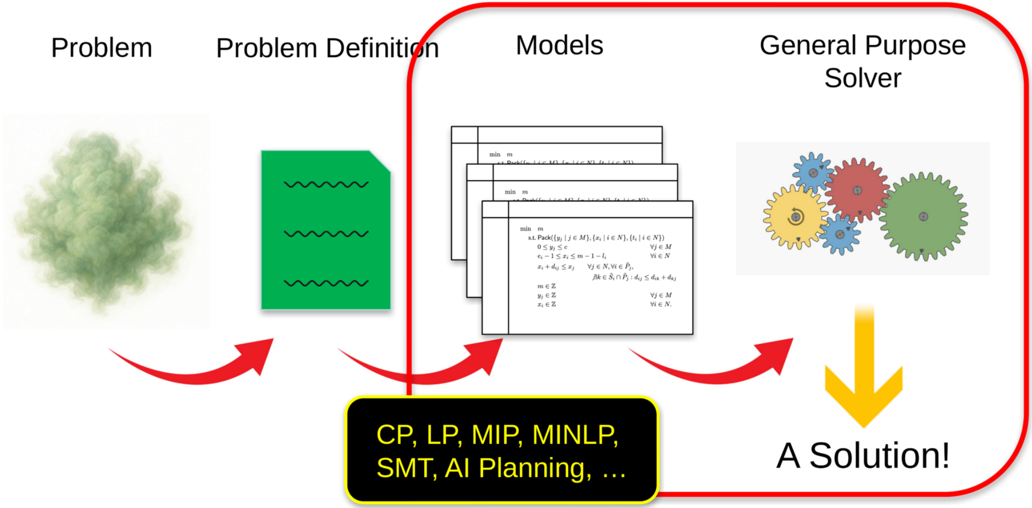
1. Trial and error on paper
2. Write a custom search algorithm
3. Reuse an existing, generic problem solving paradigm $\Leftarrow$ **This Course**

Model-and-Solve

Problem Problem Definition



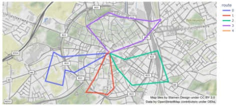
Model-and-Solve

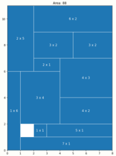


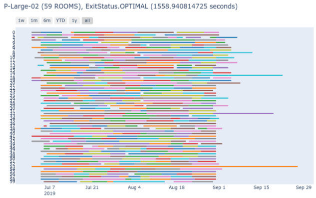
Outline

Outline

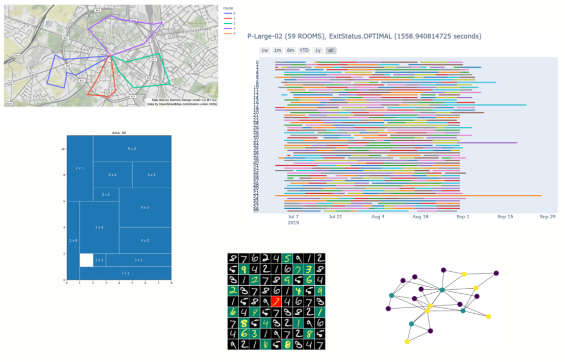
Combinatorial Optimisation







Combinatorial Optimisation



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

Example (Agricultural experiment design)

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

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

Example (Doctor rostering)

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

Constraints to be **satisfied**:

1. #on-call doctors / day $= 1$
2. #operating doctors / weekday ≤ 2
3. #operating doctors / week ≥ 7
4. #appointed doctors / 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	appt	call	none	oper	none	none	call
Doctor C	oper	none	call	appt	appt	call	none
Doctor D	appt	oper	none	call	oper	none	none
Doctor E	oper	none	oper	none	call	none	none

Constraints to be **satisfied**:

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

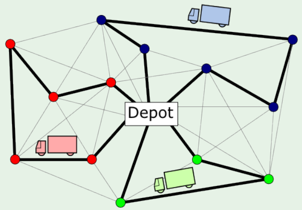
Objective function to be **minimised**: Cost: ...

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**:

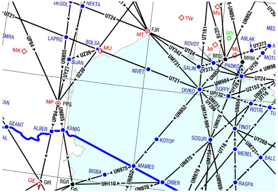
- Cost: the total fuel consumption and driver salary.

Example (Travelling salesperson: optimisation TSP)

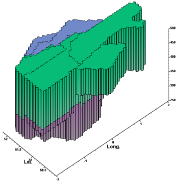
Given a map and cities,
find a **shortest** route visiting each city once and returning to the starting city.

Applications in Air Traffic Management

Demand vs capacity



Airspace sectorisation



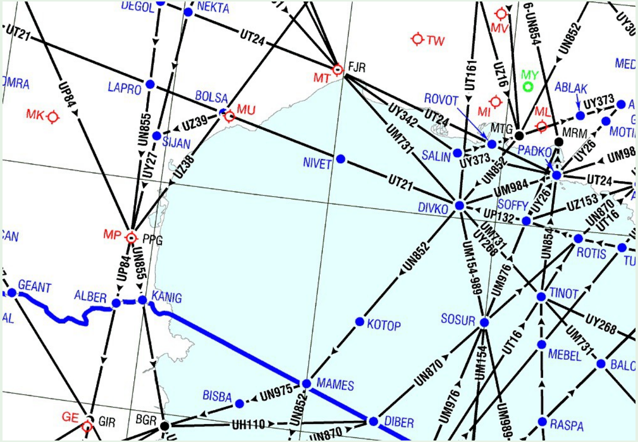
Contingency planning

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)

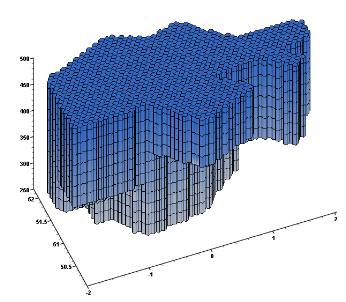
Reroute flights, in height and speed, so as to balance the workload of air traffic controllers in a multi-sector airspace:

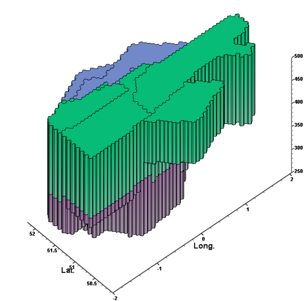


Example (Airspace sectorisation)

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

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

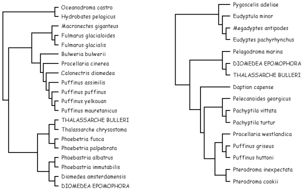




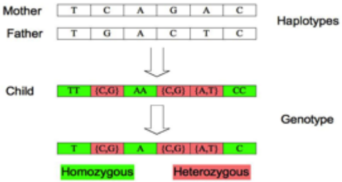
There are s^c possible colourings, but very few optimally satisfy the constraints: is intelligent search necessary?

Applications in Biology and Medicine

Phylogenetic supertree



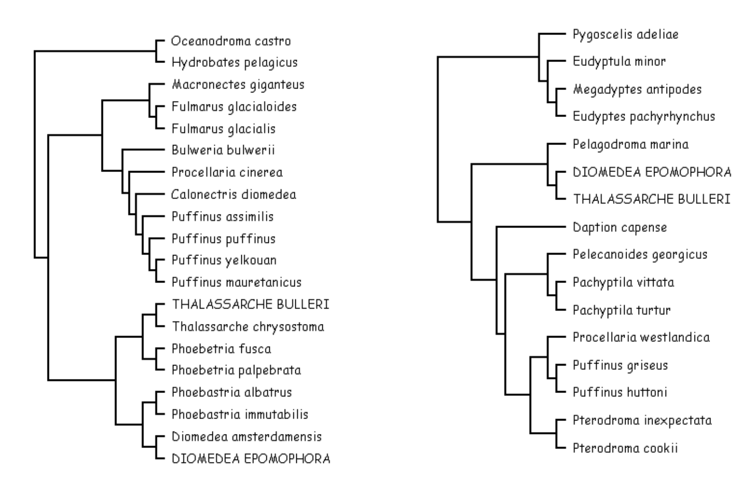
Haplotype inference



Medical image analysis

Doctor rostering

Example (Given several phylogentic trees, what supertree is maximally consistent with shared species in the trees?)



Example (Haplotype inference by pure parsimony)

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

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

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

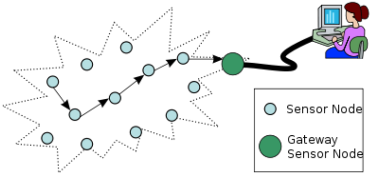
...					
A	C	T	C	T	C
...					
A	G	T	C	A	C
...					
T	G	T	G	A	C
...					

so that each given genotype conflates (is the merge of) 2 found haplotypes.

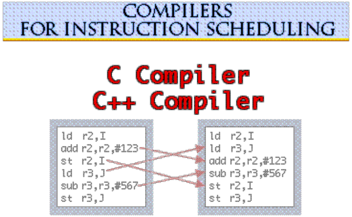
Applications in Programming and Testing

Robot programming

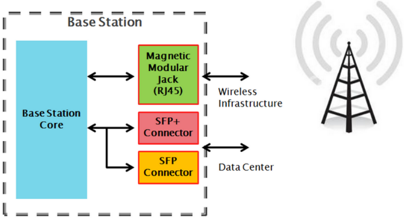
Sensor-net configuration



Compiler design

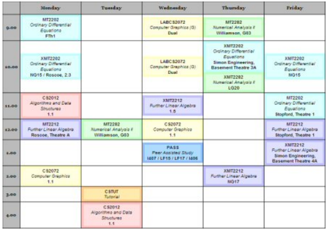


Base-station testing



Other Application Areas

School timetabling



	Monday	Tuesday	Wednesday	Thursday	Friday
9.00	MT2202 *Ordinary Differential Equations* FTh1		LABCS2072 *Computer Graphics (G)* Dual	MT2282 *Numerical Analysis II* Williamson, G03	
10.00	XMT2202 *Ordinary Differential Equations* NG15 / Roscoe, 2.3		LABCS2072 *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

Outline

Let's reconsider:

Example (Agricultural experiment design)

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

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

Could you compute a solution for 50 grains and 20 plots? for 1000 grains?

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

Informally:

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

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

Problems that are solvable in polynomial time (in the input size) are considered tractable, aka easy.
Problems needing super-polynomial time are considered intractable, aka hard.

NP Completeness: Examples

Given a digraph (V, E):

Examples

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

NP Completeness: More Examples

Examples

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

Search spaces are often larger than the universe!

Many important real-life problems are NP-hard or worse: their real-life instances can only be solved exactly and fast enough by intelligent search, unless $\mathrm{P} = \mathrm{NP}$.

NP-hardness is not where the fun ends, its where it begins!

Example (Optimisation TSP over n cities)

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

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

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

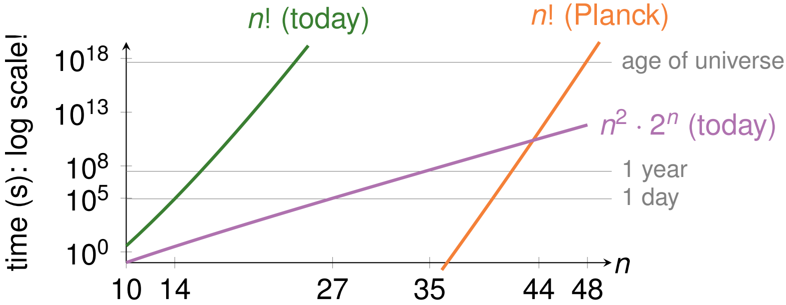
- Planck time is shortest useful interval: $\approx 5.4 \cdot 10^{-44}$ second; a Planck computer would evaluate $1.8 \cdot 10^{43}$ routes / second:

n	time
37	0.7 seconds
41	20 days
48	$1.5 \cdot$ 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 beats the $\mathcal{O}(n!)$ algo on Planck computer for $n \geq 44$.

Intelligent Search upon NP-Hardness

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



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

A declarative problem solving paradigm offers languages, methods, and tools for:

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

and / or

how: **Solving** combinatorial problems intelligently:

- Search: Explore the space of possible assignments.
- Inference: Reduce the space to feasible (partial) assignments.
- Relaxation: Exploit solutions to problems with fewer or simplified constraints.

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

The ideas in this course extend to continuous optimisation, stochastic optimisation, planning and more

Examples (Declarative problem solving paradigms)

General-purpose solvers, taking model and data as input:

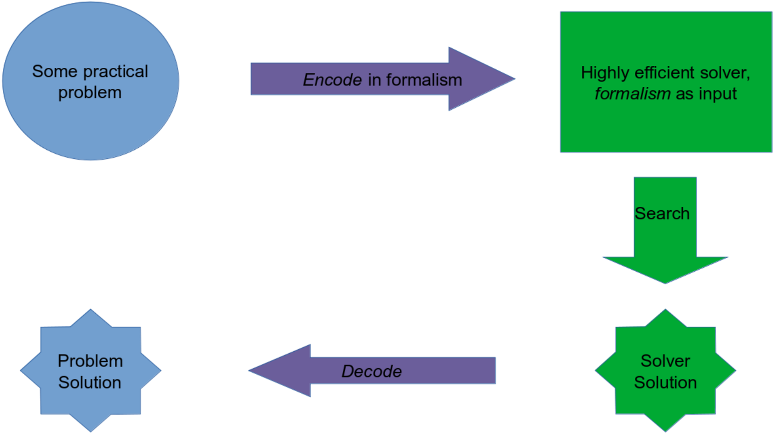
- SAT: Boolean satisfiability
- PB: Pseudo-Boolean Optimisation (0-1 linear constraints)
- SMT/OMT: SAT (resp. Optimisation) Modulo Theories
- MIP: Mixed Integer (Linear) Programming
- CP: Constraint programming
- …

Examples (Solving methodologies)

Methodologies (typically without separated concept of ’model’ and ’solver’):

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

Solvers in AI



Solvers in AI

Map Coloring

Some practical problem

Encode in formalism

CSP

Highly efficient solver, *formalism* as input

CSP solver

Solver does backtracking with AC3 and value/var heuristics

Solver Solution

WA=1, NT=2, SA=3, Q=1, NSW=2, V=1, T=3

Decode

Problem Solution

Examples:
- CSP with CSP solver
- CNF with SAT solver (this lecture)
- STRIPS with Planning solver (next lecture)
- First Order Logic with Prolog solver
- Mixed Integer Programming with MIP solver
- ...

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:

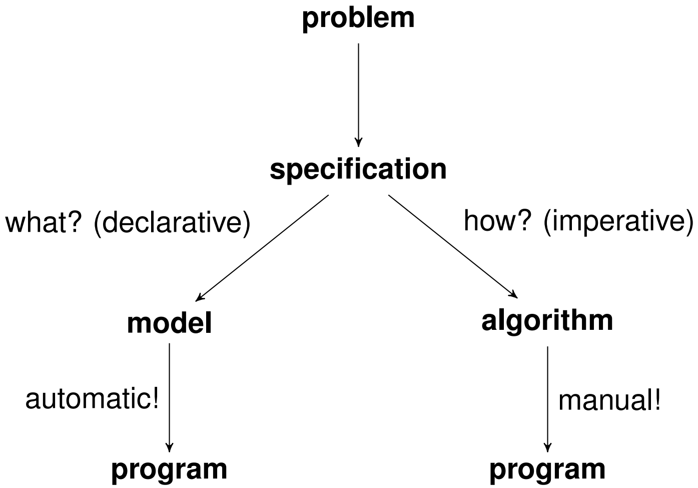
$$\text{sort}(A, S) \equiv \text{permutation}(A, S) \wedge \text{increasing}(S)$$

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

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

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

Modelling vs Programming



Definitions

A combinatorial optimisation problem consists of:

- **Decision variables**: the unknowns for which values have to be found
- **Domains**: for each variable what its allowed values are
- **Constraints**: relations between decision variables that must be satisfied
- optionally an **Objective function**: a mathematical function over the decision variables that must be minimized or maximized.

Definitions

An assignment maps each decision variable to a value within its domain; it is:

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

The search space consists of all possible assignments.
A solution to a satisfaction problem is a feasible assignment.
An optimal solution to an optimisation problem is a feasible & optimal assignment.

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

The goal of Sudoku is to *complete* a 9x9 grid with numbers so that each row, column and 3x3 section contain each digit between 1 and 9 once.

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

Example (Sudoku CP model)

$$G_{ij} \in \{1, 2, \ldots, 9\}, \qquad \forall i, j \in \{1, 2, \ldots, 9\} \tag{1}$$

$$\textsc{AllDifferent}(G_{i1}, G_{i2}, \ldots, G_{i9}), \qquad \forall i \in \{1, 2, \ldots, 9\} \tag{2}$$

$$\textsc{AllDifferent}(G_{1j}, G_{2j}, \ldots, G_{9j}), \qquad \forall j \in \{1, 2, \ldots, 9\} \tag{3}$$

$$\begin{aligned} \textsc{AllDifferent}(G_{p,q},\quad & G_{p,q+1},\quad G_{p,q+2}, \\ & G_{p+1,q}, G_{p+1,q+1}, G_{p+1,q+2}, \\ & G_{p+2,q}, G_{p+2,q+1}, G_{p+2,q+2}), \end{aligned} \qquad \forall p, q \in \{1, 4, 7\} \tag{4}$$

$$G_{ij} = \text{given}_{ij}, \qquad \text{if a value given}_{ij} \text{ is given} \tag{5}$$

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

Example (Sudoku in CPMpy (indexing offset 0))

```python
import cpmpy as cp
#given = np.array(...)  # load the hints, uses '0' for the empty cells
grid = cp.intvar(1,9, shape=given.shape, name="grid")  # Decision variables
model = cp.Model(
    [cp.AllDifferent(row) for row in grid],
    [cp.AllDifferent(col) for col in grid.T],  # numpy's Transpose
    [cp.AllDifferent(grid[i:i+3, j:j+3]) \
        for i in range(0, 9, 3) for j in range(0, 9, 3)],
    grid[given!=0] == given[given!=0], # enforce the hints
)
model.solve()
```

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

Example (Sudoku in MiniZinc (indexing offset 1))

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

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.

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 problem: balanced incomplete block design (BIBD)

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

Example (BIBD *integer* CP model)

$$
\begin{aligned}
& v = 7, \quad b = 7, && \text{Varieties, Blocks} && (6)\\
& k = 3, \quad r = 3, \quad \lambda = 1, && \text{sample size, block size, balance} && (7)\\
& && && (8)\\
& B_{ij} \in \{0,1\}, && \forall i \in \{1,2,\ldots,v\}, \forall j \in \{1,2,\ldots,b\}, && (9)\\
& \sum_{j=1}^{b} B_{ij} = k, && \forall i \in \{1,2,\ldots,v\}, \text{every row must add up to sample size} && (10)\\
& \sum_{i=1}^{v} B_{ij} = r, && \forall j \in \{1,2,\ldots,b\}, \text{every columns must add up to block size} && (11)\\
& \sum_{j=1}^{b} B_{ij} B_{i'j} = \lambda, && \forall i, i' \in \{1,2,\ldots,v\},\ i \neq i'. \text{every distinct row, scalar product = balance} && \\
& && && (12)
\end{aligned}
$$

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

Example (BIBD *integer* model in CPMpy)

```
varieties,blocks = 7,7
sampleSize,blockSize = 3,3
balance = 1

BIBD = cp.boolvar(shape=(varieties, blocks),name="matrix")

model = cp.Model(
    # every row must add up to sampleSize
    [cp.sum(row) == sampleSize for row in BIBD],
    # every column must add up to blocksize
    [cp.sum(col) == blockSize for col in BIBD.T],
    # the scalar product of every pair of distinct rows must sum up to balance
    [cp.sum(row_i*row_j) == balance for row_i, row_j in all_pairs(BIBD)]
)

model.solve()
```

Reconsider the model fragment:

```
    [cp.sum(row) == sampleSize for row in BIBD]
```

This constraint is declarative,
so read it using only the verb “must be” or similar:

> *for each row* `row` *of* `BIBD`*,*
> *the sum of values in row* `row`
> *must be equal to* `sampleSize`

The constraint is NOT procedural:

> *for each row* `row` *of* `BIBD`*,*
> *we first sum the values in row* `row`
> *and then check if that count equals* `sampleSize -- !WRONG!`

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

Symbolic model creation in Python

Declarative programming in a procedural language???

```
[cp.sum(row) == sampleSize for row in BIBD]
```

Yes, this is a declarative specification: the sum and comparison are not *executed*, instead they *create objects*!

The result is a list of CPMpy `Expression` objects.

These expressions are passed *symbolically* to a solver, who will create a search space and (proceduraly) search for a solution to all constraints.

```
m = cp.Model([cp.sum(row) == sampleSize for row in BIBD])
print(m)
m.solve()
```

Example (BIBD *set-based* representation)

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.

Decision variables are a choice:
we could model the same problem with one *set* variable per grain.

Decision variables are a choice:
we could model the same problem with one *set* variable per grain variety.

Example (BIBD *set* CP model)

$$v = 7, \quad b = 7, \qquad \text{Varieties, Blocks} \tag{13}$$

$$k = 3, \quad r = 3, \quad \lambda = 1, \qquad \text{sample size, block size, balance} \tag{14}$$

$$\tag{15}$$

$$\mathcal{B}_i \subseteq \{1, 2, \ldots, b\}, \qquad \forall i \in \{1, 2, \ldots, v\}, \text{ Set of blocks for each variety} \tag{16}$$

$$\sum_{i=1}^{v} [j \in \mathcal{B}_i] = r, \qquad \forall j \in \{1, 2, \ldots, b\}, \text{ Each block contains exactly block-size varieties} \tag{17}$$

$$|\mathcal{B}_i| = k, \qquad \forall i \in \{1, 2, \ldots, v\}, \text{ Each variety appears in exactly sample-size blocks} \tag{18}$$

$$|\mathcal{B}_i \cap \mathcal{B}_j| = \lambda, \qquad \forall i, j \in \{1, 2, \ldots, v\}, \ i \neq j, \text{ balance between each pair of variaties} \tag{19}$$

Note: not all modeling languages support *set* decision variables.
MiniZinc does, CPMpy does not.

Example (Doctor rostering)

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

Constraints to be **satisfied**:

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

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

Example (Doctor Shift Scheduling CP Model)

$$R_{pd} \in \{0, 1, \ldots, \text{n\_shifts} - 1\}, \qquad \forall p \in \{1, \ldots, \text{n\_doctors}\},\ d \in \{1, \ldots, \text{n\_days}\} \tag{20}$$

1. #on-call doctors / day = 1: $$\sum_{p=1}^{\text{n\_doctors}} [R_{pd} = \text{Call}] = 1, \qquad \forall d \in \{1, \ldots, \text{n\_days}\} \tag{21}$$

2. #operating doctors / weekday <= 2: $$\sum_{p=1}^{\text{n\_doctors}} [R_{pd} = \text{Oper}] \leq 2, \qquad \forall d \in \{1, \ldots, \text{n\_days}\}, \text{ if } d \mod 7 \leq 5 \tag{22}$$

3. #operating doctors / week $\geq$ 7: $$\sum_{p=1}^{\text{n\_doctors}} \sum_{d=s}^{s+6} [R_{pd} = \text{Oper}] \geq 7, \qquad \forall s \in \{1, \ldots, \text{n\_days}\}, \text{ if } d \mod 7 == 0 \tag{23}$$

4. #appointed doctors / week $\geq$ 4: $$\sum_{p=1}^{\text{n\_doctors}} \sum_{d=s}^{s+6} [R_{pd} = \text{Appt}] \geq 4, \qquad \forall s \in \{1, \ldots, \text{n\_days}\}, \text{ if } d \mod 7 == 0 \tag{24}$$

5. day off after operation: $$[R_{pd} = \text{Oper}] \rightarrow [R_{p,d+1} = \text{Free}], \qquad \forall p \in \{1, \ldots, \text{n\_doctors}\},\ d \in \{1, \ldots, \text{n\_days} - 1\} \tag{25}$$

Example (Data for our doctor rostering)

```
n_days = 7; n_doctors = 5
n_shifts = 4; Appt, Call, Oper, Free = range(n_shifts)
```

Example (CPMpy model for our doctor rostering (indexing offset 0))

```
roster = cp.intvar(0,n_shifts-1, shape=(n_doctors,n_days))
model  = cp.Model(
    # on-call/day = 1
    [cp.Count(roster[:,d], Call) == 1 for d in range(n_days)],
    # oper/weekday <= 2; assume d mod 7 == 0 for Monday, etc
    [cp.Count(roster[:,d], Oper) <= 2 for d in range(n_days) if d % 7 <= 4],
    # oper/week >= 7
    [cp.Count(roster[:,s:s+7], Oper) >= 7 for s in range(0, n_days, 7)],
    # appt/week >= 4
    [cp.Count(roster[:,s:s+7], Appt) >= 4 for s in range(0, n_days, 7)],
    # free after oper
    [(roster[p,d] == Oper).implies(roster[p,d+1] == Free) for p in range(n_doctors
    ) for d in range(n_days-1)],
)
# maximize nr of free shifts in weekend
model.maximize(cp.sum([cp.Count(roster[:,s+5:s+7], Free) for s in range(0, n_days,
     7)]))
model.solve()
```

Example (Job allocation at minimal salary cost)

Given *n_jobs* jobs and the salaries of work applicants *salary*,
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:

$$\text{salary}_a = \textit{...given value...} \qquad \forall a \in \{1, \ldots, \text{n\_apps}\} \tag{26}$$

$$W_j \in \{1, \ldots, \text{n\_apps}\}, \qquad \forall j \in \{1, \ldots, \text{n\_jobs}\} \tag{27}$$

$$\text{...constraints...} \tag{28}$$

$$\tag{29}$$

$$\text{minimize} \sum_{j=1}^{\text{n\_jobs}} \text{salary}_{W_j} \qquad \textit{observe: indexing with a decision variable!} \tag{30}$$

Example (Job allocation at minimal salary cost)

Given *n_jobs* jobs and the salaries of work applicants *salary*,
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:

```python
# n_apps = ..., n_jobs = ..., salary = ...
worker = cp.intvar(0, n_apps-1, shape=n_jobs)  # an applicant per job

model  = cp.Model(
            # qualifications, workload, etc
         )

salary = cp.cpm_array(salary)  # make it indexible by variables
model.minimize(cp.sum([salary[worker[j]] for j in range(n_jobs)]))
```

Special power of constraint programming languages:

Using a decision variable (`worker[j]`) as an index into an array (`salary[]`)

Internally: will use an 'Element' global constraint

Example (Traveling Salesperson Problem)

$$\textit{distance} = \begin{bmatrix} 0 & 85 & 162 & 231 \\ 85 & 0 & 98 & 128 \\ 162 & 98 & 0 & 146 \\ 231 & 128 & 146 & 0 \end{bmatrix} \tag{31}$$

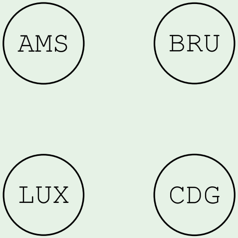
$$\textit{n\_cities} = 4 \tag{32}$$

$$\textit{Next} \in \{0, \ldots, \textit{n\_cities} - 1\} \tag{33}$$

$$\text{minimize} \sum_{c=1}^{\text{n\_cities}} \text{distance}[c, \textit{Next}_c] \tag{34}$$

(35)

(36)



	AMS	BRU	LUX	CDG
Next:				

How to ensure that *Next* represents a tour?

Example (Traveling Salesperson Problem)

$$distance = \begin{bmatrix} 0 & 85 & 162 & 231 \\ 85 & 0 & 98 & 128 \\ 162 & 98 & 0 & 146 \\ 231 & 128 & 146 & 0 \end{bmatrix} \tag{31}$$

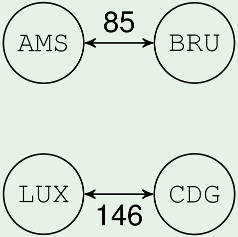
$$n\_cities = 4 \tag{32}$$

$$Next \in \{0, \ldots, n\_cities - 1\} \tag{33}$$

$$\text{minimize} \sum_{c=1}^{\text{n\_cities}} \text{distance}[c, Next_c] \tag{34}$$

$$\textsc{AllDifferent}(Next) \tag{35}$$

(36)



	AMS	BRU	LUX	CDG
Next:	BRU	AMS	CDG	LUX

So ALLDIFFERENT(*Next*) is too weak, it does not ensure *one* tour.

Example (Traveling Salesperson Problem)

$$distance = \begin{bmatrix} 0 & 85 & 162 & 231 \\ 85 & 0 & 98 & 128 \\ 162 & 98 & 0 & 146 \\ 231 & 128 & 146 & 0 \end{bmatrix} \tag{31}$$

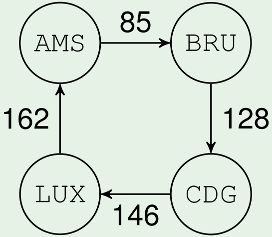
$$n\_cities = 4 \tag{32}$$

$$Next \in \{0, \ldots, n\_cities - 1\} \tag{33}$$

$$\text{minimize} \sum_{c=1}^{\text{n\_cities}} \text{distance}[c, Next_c] \tag{34}$$

$$\textsc{Circuit}(Next) \tag{35}$$

(36)



	AMS	BRU	LUX	CDG
Next:	BRU	CDG	AMS	LUX

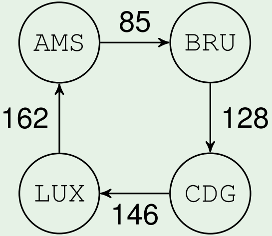
For this, the CIRCUIT() global constraint is needed instead!

Example (Traveling Salesperson Problem)

```
n_cities = 4;
distance = cp.cpm_array([
    [0,    85, 162, 231],    # Km from AMS
    [85,    0,  98, 128],    # Km from BRU
    [162,  98,   0, 146],    # Km from LUX
    [231, 128, 146, 0  ]     # Km from CDG
])
# Travel from c to Next[c]
Next = cp.intvar(0, n_cities-1, shape=
    n_cities)

# Successor variables must from a circuit
model = cp.Model( cp.Circuit(Next) )

model.minimize(cp.sum(distance[c, Next[c]]
    for c in range(n_cities)))
```



	AMS	BRU	LUX	CDG
Next:	BRU	CDG	AMS	LUX

Special power of constraint programming languages:

Using a decision variable (`Next[c]`) as an index into an array (`distance[]`)

Decision Variables, Parameters, and Identifiers

- Decision variables and parameters in a model are concepts very different from programming variables in an imperative or object-oriented program.
- A decision variable in a model is like a variable in mathematics: it is *not* given a value in a model or a formula, and its value is only fixed in a solution, if a solution exists.
- A parameter in a model must be given a value, but only once: we say that the parameter is instantiated.
- A decision variable or parameter is referred to by an identifier (a name like `BIBD`).
- An index identifier in a universal quantification takes on all its designated values in turn. Example: $\forall i \in \{1, 2, \ldots, v\}$ where i is an index identifier and v is a parameter.

Parameterised Constraint Models

A constraint model, e.g. a constraint specification, also written as simply *model* in this course, is typically written down as a parameterised constraint model.

- A parameterized constraint model has uninstantiated parameters. For example the sample size k in BIBD's $\sum_{j=1}^{b} B_{ij} = k$ can be different for different problem instances.
- The parameters correspond to the input data. For example the number of grains and plots, and k/sample size in the BIBD problem; or the stops and distance matrix in a vehicle routing problem.
- An instance is the combination of data and a parametrized constraint model, the instance is the result of instantiating the parameters, and in turn all the decision variables and constraints, with the data.

Modelling Concepts (end)

- A constraint is a restriction on the values that its decision variables can take together; equivalently, it is a Boolean-valued expression over decision variables that is asserted to be true.
- An objective function is a numeric expression over decision variables whose value is to be either minimised or maximised.
- Finally, we can ask a solver to **compute** different things:
  - find any satisfying solution
  - find all satisfying solutions
  - find an optimal solutions
  - find all optimal solution
  - count the number of satisfying/optimal solutions
  - prove that there is no solution
  - find a minimal subset of unsatisfiable constraints
  - …

Constraint-Based Modelling

CPMpy is a Python-based constraint modelling library (*not* a solver):

- Decision variables are n-dimensional **numpy arrays**, and you can specify constraints using standard Python and NumPy functions.
  only *Boolean* and *integer* decision variables are possible.
- Standard Python operators (+ - / & | `sum`() `abs`() `min`() `max`()) and comparisons (== >= > != <= <) can be used, and there is a large library of global constraints (`AllDifferent`(), `Circuit`(), `Cumulative`(), ...) and global functions (`Count`(), `Element`(), ...)
- There is support for both constraint satisfaction, optimisation (solve()) and solution counting/enumeration (solveAll()).

Compared to the library of a specific solver (e.g. *ortools* or *gurobi*), you can specify problems at a higher level, e.g. nested expressions, and use globals that the solver might not support. CPMpy will translate this to a lower-level solver library for you.

Correctness Is Not Enough for Models

Modelling is a craft!

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

Good modellers are highly valued in industry!

Use solvers: based on decades of cutting-edge research,
they are very hard to beat in finding optimal solutions.

Outline

Solving a model

MiniZinc and CPMpy are solver-independent modelling frameworks: they can translate to multiple different solvers.

Expressiveness of key declarative solving paradigms:

- **SAT:** Boolean decision variables; clauses as constraints
- **LP:** Floating-point decision variables; linear constraints & objective
- **MIP:** Floating-point&Integer decision variables; linear constraints & objective
- **CP:** Bool&Int decision variables; logical, mathematical, global constraints
- **SMT:** Bool&Int&String&... decision variables; logical, theory-specific

There Are Many Solving Technologies

- No technology universally dominates all the others
- One should test several technologies on each problem
- Some technologies have standardised modelling languages across all solvers: SAT, PseudoBoolean, ILP/MIP, SMT
- Some technologies have non-standardised modelling languages across their solvers: CP and LCG *(although: XCSP3)*
- Some technologies even have no modelling languages: local search, dynamic prog., and genetic algorithms are rather methodologies.

How to Solve a Combinatorial Optimisation Problem?

1. **Model the problem**

2. **Have a solver solve it**

Easy, right?

How to Solve a Combinatorial Optimisation Problem?

1. **Model the problem**
   - Understand the problem
   - Choose the decision variables and their domains
   - Formulate the constraints
   - Formulate the objective function, if any
   - Make sure the model really represents the problem; iterate

2. **Have a solver solve it**
   - Choose a solving technology and solver
   - Choose the hyper-parameters, potentially the search strategy
   - Run the model and interpret the (lack of) solution(s)
   - Debug or improve the model, if need be; iterate

Not so easy, but easier than implementing combinatorial algorithms from scratch!

Model and Solve

Advantages:

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

Disadvantages:

– Do I need to learn several modelling languages? Not in this course!

– Do I need to understand the used solving technologies in order to get the most out of them? Yes, but . . . !

Outline

Course setup at KU Leuven

Tour of the online learning platform...