Chapter 00: Introduction
(Version of 20th January 2017)

Pierre Flener

Department of Information Technology
Computing Science Division
Uppsala University
Sweden

Course 1DL481:
Algorithms and Data Structures 3
Outline

1. The End of Course AD2
2. Combinatorial Optimisation
   - Constraint Problems
   - Solving Technologies
   - Modelling
   - Solvers
3. Course AD3
   - Contents
   - New Concepts
   - Learning Outcomes
   - Organisation
In a decision problem we seek a ‘yes’ / ‘no’ answer to an existence question. An instance is given by its input data.

**Example (Travelling salesperson: Decision TSP)**

*Given* a map and *n* cities, *is* there a route visiting each city once, returning to the starting city, and costing at most *c*?
In a decision problem we seek a ‘yes’ / ‘no’ answer to an existence question. An instance is given by its input data.

**Example (Travelling salesperson: Decision TSP)**

Given a map and \( n \) cities, is there a route visiting each city once, returning to the starting city, and costing at most \( c \)?

A decision problem \( R \) is:

- in NP if a witness to a ‘yes’ instance is checkable in time polynomial in the instance size: checking is in \( P \);
In a decision problem we seek a ‘yes’ / ‘no’ answer to an existence question. An instance is given by its input data.

**Example (Travelling salesperson: Decision TSP)**

*Given* a map and *n* cities, *is* there a route visiting each city once, returning to the starting city, and costing at most *c*?

A decision problem *R* is:
- in NP if a witness to a ‘yes’ instance is checkable in time polynomial in the instance size: checking is in P;
- NP-complete if it is in NP and there is a reduction from each problem *Q* in NP, transforming in polytime each instance of *Q* into an instance of *R* of the same answer.

It is believed that NP-complete problems are intractable (or: hard), requiring super-polynomial time to solve exactly.
In a decision problem we seek a ‘yes’ / ‘no’ answer to an existence question. An instance is given by its input data.

**Example (Travelling salesperson: Decision TSP)**

Given a map and \( n \) cities, is there a route visiting each city once, returning to the starting city, and costing at most \( c \)?

A decision problem \( R \) is:
- **in NP** if a witness to a ‘yes’ instance is checkable in time polynomial in the instance size: checking is in \( P \);
- **NP-complete** if it is in NP and there is a reduction from each problem \( Q \) in NP, transforming in polytime each instance of \( Q \) into an instance of \( R \) of the same answer.

It is believed that NP-complete problems are **intractable** (or: **hard**), requiring super-polynomial time to solve **exactly**.

**Example**

TSP is NP-complete as a witness is checkable in \( O(n) \) time and NP-complete Hamiltonian-Cycle problem reduces to it.
In a **satisfaction problem** we seek a witness for ‘yes’ answer.

**Example (Satisfaction TSP)**

**Given** a map and $n$ cities, **find** a route visiting each city once, returning to the starting city, and costing at most $c$. 
In a satisfaction problem we seek a witness for ‘yes’ answer.

Example (Satisfaction TSP)

**Given** a map and \( n \) cities, **find** a route visiting each city once, returning to the starting city, and costing at most \( c \).

In an optimisation problem we seek an optimal witness according to some objective function for a ‘yes’ answer.

Example (Optimisation TSP)

**Given** a map and \( n \) cities, **find** a **cheapest** route visiting each city once and returning to the starting city.
In a **satisfaction problem** we seek a witness for ‘yes’ answer.

**Example (Satisfaction TSP)**

**Given** a map and \( n \) cities, **find** a route visiting each city once, returning to the starting city, and costing at most \( c \).

In an **optimisation problem** we seek an optimal witness according to some **objective function** for a ‘yes’ answer.

**Example (Optimisation TSP)**

**Given** a map and \( n \) cities, **find a cheapest** route visiting each city once and returning to the starting city.

In addition to decision problems that are at least as hard as every NP problem (as every NP problem reduces to them), satisfaction and optimisation problems with NP-complete decision versions are often also said to be **NP-hard**: they are unlikely to be easier than their decision versions.
What Now?

Several courses at Uppsala University teach techniques for addressing NP-hard optimisation and satisfaction problems:

1TD184  Continuous Optimisation  (period 2)

1DL449  Modelling for Combinatorial Optimisation  (period 3)

1DL441  Comb’l Opti. with Constraint Programming  (period 1+2)

1DL481  Algorithms and Data Structures 3  (period 3)
What Now?

Several courses at Uppsala University teach techniques for addressing NP-hard optimisation and satisfaction problems:

1TD184  Continuous Optimisation  (period 2)

1DL449  Modelling for Combinatorial Optimisation  (period 3)

1DL441  Comb’l Opti. with Constraint Programming  (period 1+2)

1DL481  Algorithms and Data Structures 3  (period 3)

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:

$$
\begin{array}{c|c}
 n & \text{time} \\
11 & \\
\end{array}
$$

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:

$$
\begin{array}{c|c}
 n & \text{time} \\
37 & 0.7 \text{ seconds} \\
41 & 20 \text{ days} \\
48 & 1.5 \cdot \text{age of universe} \\
\end{array}
$$

The dynamic program by Bellman-Held-Karp "only" takes $O(n^2 \cdot 2^n)$ time: a computer of today takes a day for $n=27$, a year for $n=35$, the age of the universe for $n=67$, and it beats $n!$ on the Planck computer for $n \geq 44.$
## Example (Optimisation TSP over $n$ cities)

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>40 seconds</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.7</td>
<td>41 days</td>
</tr>
<tr>
<td>1.5 $\cdot$ age of universe</td>
<td></td>
</tr>
</tbody>
</table>

The dynamic program by Bellman-Held-Karp "only" takes $O(n^2 \cdot 2^n)$ time: a computer of today takes a day for $n = 27$, a year for $n = 35$, the age of the universe for $n = 67$, and it beats $n!$ on the Planck computer for $n \geq 44$. 
Example (Optimisation TSP over $n$ cities)

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>40 seconds</td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
Example (Optimisation TSP over $n$ cities)

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>40 seconds</td>
</tr>
<tr>
<td>14</td>
<td>1 day</td>
</tr>
</tbody>
</table>

The dynamic program by Bellman-Held-Karp "only" takes $O(n^2 \cdot 2^n)$ time: a computer of today takes a day for $n=27$, a year for $n=35$, the age of the universe for $n=67$, and it beats $n!$ on the Planck computer for $n \geq 44$. 
Example (Optimisation TSP over $n$ cities)

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>40 seconds</td>
</tr>
<tr>
<td>14</td>
<td>1 day</td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>
Example (Optimisation TSP over $n$ cities)

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>40 seconds</td>
</tr>
<tr>
<td>14</td>
<td>1 day</td>
</tr>
<tr>
<td>18</td>
<td>203 years</td>
</tr>
</tbody>
</table>

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

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

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

The End of Course AD2

Combinatorial Optimisation
Constraint Problems
Solving Technologies
Modelling
Solvers

Course AD3
Contents
New Concepts
Learning Outcomes
Organisation

Course 1DL481
Example (Optimisation TSP over \( n \) cities)

A brute-force algorithm evaluates all \( n! \) candidate routes:

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

\[
\begin{array}{cc}
n & \text{time} \\
11 & 40 \text{ seconds} \\
14 & 1 \text{ day} \\
18 & 203 \text{ years} \\
20 &
\end{array}
\]
Example (Optimisation TSP over \(n\) cities)

A brute-force algorithm evaluates all \(n!\) candidate routes:

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

<table>
<thead>
<tr>
<th>(n)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>40 seconds</td>
</tr>
<tr>
<td>14</td>
<td>1 day</td>
</tr>
<tr>
<td>18</td>
<td>203 years</td>
</tr>
<tr>
<td>20</td>
<td>77k years</td>
</tr>
</tbody>
</table>
### Example (Optimisation TSP over $n$ cities)

A brute-force algorithm evaluates all $n!$ candidate routes:
- A computer of today evaluates $10^6$ routes / second:
  
<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>40 seconds</td>
</tr>
<tr>
<td>14</td>
<td>1 day</td>
</tr>
<tr>
<td>18</td>
<td>203 years</td>
</tr>
<tr>
<td>20</td>
<td>77k years</td>
</tr>
</tbody>
</table>

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

A brute-force algorithm evaluates all \( n! \) candidate routes:

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

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

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>370.7 seconds</td>
</tr>
</tbody>
</table>

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

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

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

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>0.7 seconds</td>
</tr>
</tbody>
</table>
Example (Optimisation TSP over \( n \) cities)

A brute-force algorithm evaluates all \( n! \) candidate routes:

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

\[
\begin{array}{c|c}
 n & \text{time} \\
\hline
 11 & 40 \text{ seconds} \\
 14 & 1 \text{ day} \\
 18 & 203 \text{ years} \\
 20 & 77k \text{ years} \\
\end{array}
\]

- 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:

\[
\begin{array}{c|c}
 n & \text{time} \\
\hline
 37 & 0.7 \text{ seconds} \\
 41 & \\
\end{array}
\]
Example (Optimisation TSP over $n$ cities)

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

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

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>0.7 seconds</td>
</tr>
<tr>
<td>41</td>
<td>20 days</td>
</tr>
</tbody>
</table>
Example (Optimisation TSP over $n$ cities)

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

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

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>0.7 seconds</td>
</tr>
<tr>
<td>41</td>
<td>20 days</td>
</tr>
<tr>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>
Example (Optimisation TSP over $n$ cities)

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

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

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

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

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

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

- A computer of today evaluates $10^6$ routes / second:
  
<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>40 seconds</td>
</tr>
<tr>
<td>14</td>
<td>1 day</td>
</tr>
<tr>
<td>18</td>
<td>203 years</td>
</tr>
<tr>
<td>20</td>
<td>77k years</td>
</tr>
</tbody>
</table>

- Planck time is shortest useful interval: $\approx 5.4 \cdot 10^{-44}$ s; 
a Planck computer would evaluate $1.8 \cdot 10^{43}$ routes / s:
  
<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>0.7 seconds</td>
</tr>
<tr>
<td>41</td>
<td>20 days</td>
</tr>
<tr>
<td>48</td>
<td>1.5 $\cdot$ age of universe</td>
</tr>
</tbody>
</table>

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

$n!$ (today)

$n!$ (Planck)

age of universe

$n^2 \cdot 2^n$ (today)

1 year

1 day

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

$n!$ (today)

$n!$ (Planck)

age of universe

Concorde TSP Solver beats the Bellman-Held-Karp exact algo: it uses approximation and stochastic local search algos, but it can sometimes prove the optimality of its solutions. The largest instance it has solved exactly, in 136 CPU years in 2006, has 85,900 cities! Let the fun begin!
Outline

1 The End of Course AD2

2 Combinatorial Optimisation
   - Constraint Problems
   - Solving Technologies
   - Modelling
   - Solvers

3 Course AD3
   - Contents
   - New Concepts
   - Learning Outcomes
   - Organisation
Optimisation is a science of *service*: to scientists, to engineers, to artists, and to society.
The End of Course AD2

Combinatorial Optimisation

Constraint Problems

Solving Technologies

Modelling

Solvers

Course AD3

Contents

New Concepts

Learning Outcomes

Organisation
### Example (Touristic town competition)

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Birka</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Falun</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lund</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mora</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sigtuna</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uppsala</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Västerås</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Constraints to be satisfied:**

- Equal jury size: Every town is evaluated by 3 judges.
- Equal travel load: Every judge evaluates 3 towns.
- Fairness: Every town pair has 1 judge in common.
Example (Touristic town competition)

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Birka</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Falun</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lund</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mora</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Sigtuna</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uppsala</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Västerås</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Constraints to be satisfied:
- Equal jury size: Every town is evaluated by 3 judges.
- Equal travel load: Every judge evaluates 3 towns.
- Fairness: Every town pair has 1 judge in common.
### Example (Doctor rostering)

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

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

**Objective function to be minimised:**
- Cost: . . .
Example (Doctor rostering)

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

Constraints to be satisfied:

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

Objective function to be minimised:

- Cost: . . .
Example (Vehicle routing: Parcel delivery)

Given a depot with parcels for clients, a fleet of vehicles, find which vehicle visits which clients in which order.

Constraints to be satisfied:
- All parcels are delivered on time.
- No vehicle is overloaded.
- Driver regulations are respected.
- ... 

Objective function to be minimised:
- Cost: the total fuel consumption.
The End of Course AD2
Combinatorial Optimisation
Constraint Problems
Solving Technologies
Modelling
Solvers
Course AD3
Contents
New Concepts
Learning Outcomes
Organisation

School timetabling

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 am</td>
<td>VM2120</td>
<td>LAEC25050</td>
<td>WM2120</td>
<td>VM2120</td>
</tr>
<tr>
<td>10 am</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
</tr>
<tr>
<td></td>
<td>Ordinary Differential Equations</td>
<td>Applied Mechanics</td>
<td>Ordinary Differential Equations</td>
<td>Ordinary Differential Equations</td>
</tr>
<tr>
<td>11 am</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
</tr>
<tr>
<td></td>
<td>Ordinary Differential Equations</td>
<td>Applied Mechanics</td>
<td>Ordinary Differential Equations</td>
<td>Ordinary Differential Equations</td>
</tr>
<tr>
<td>12 am</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
</tr>
<tr>
<td></td>
<td>Ordinary Differential Equations</td>
<td>Applied Mechanics</td>
<td>Ordinary Differential Equations</td>
<td>Ordinary Differential Equations</td>
</tr>
<tr>
<td>1 pm</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
</tr>
<tr>
<td></td>
<td>Ordinary Differential Equations</td>
<td>Applied Mechanics</td>
<td>Ordinary Differential Equations</td>
<td>Ordinary Differential Equations</td>
</tr>
<tr>
<td>2 pm</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
</tr>
<tr>
<td></td>
<td>Ordinary Differential Equations</td>
<td>Applied Mechanics</td>
<td>Ordinary Differential Equations</td>
<td>Ordinary Differential Equations</td>
</tr>
<tr>
<td>3 pm</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
</tr>
<tr>
<td></td>
<td>Ordinary Differential Equations</td>
<td>Applied Mechanics</td>
<td>Ordinary Differential Equations</td>
<td>Ordinary Differential Equations</td>
</tr>
<tr>
<td>4 pm</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
<td>WM2120</td>
</tr>
<tr>
<td></td>
<td>Ordinary Differential Equations</td>
<td>Applied Mechanics</td>
<td>Ordinary Differential Equations</td>
<td>Ordinary Differential Equations</td>
</tr>
</tbody>
</table>

Sports tournament design

School timetabling

Vehicle routing

Sports tournament design

Container packing

Course 1DL481

Algorithms and Data Structures 3
Robotic task sequencing

Sensor net configuration

Compiler design

Base station testing

C Compiler
C++ Compiler

1d r2,I
add r2,r2,#123
st r2,I
1d r3,I
add r2,r2,#123
sub r3,r3,#567
st r2,I
st r3,I
Many important real-life problems are NP-hard and thus must be solved by **intelligent search**, unless P = NP:

- Personnel rostering, scheduling, time-tabling, . . .
- Transportation logistics: vehicle routing, . . .
- Packing: container or truck loading, carpet cutting, . . .
- Configuration, design, experiment set-up, . . .
- Verification of hardware and software, VLSI layout, . . .
- Alignment of bio-molecules, phylogeny, . . .
- Financial investment instrument design, . . .
- . . .
Definitions

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

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

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.

A candidate solution is an assignment to all the variables of values within their domains; it is:

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

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

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

A candidate solution is an assignment to all the variables of values within their domains; it is:

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

A solution to a satisfaction problem is feasible. An optimal solution to an optimisation problem is feasible and optimal.
Definitions

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

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

A candidate solution is an assignment to all the variables of values within their domains; it is:

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

A solution to a satisfaction problem is feasible. An optimal solution to an optimisation problem is feasible and optimal.

Combinatorial optimisation covers satisfaction problems and optimisation problems, for variables over discrete sets.
Outline

1. The End of Course AD2

2. Combinatorial Optimisation
   - Constraint Problems
   - Solving Technologies
   - Modelling
   - Solvers

3. Course AD3
   - Contents
   - New Concepts
   - Learning Outcomes
   - Organisation
A solving technology offers methodologies 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**: Shrink the space of candidate solutions.
- **Relaxation**: Exploit solutions to easier problems.
Outline

1. The End of Course AD2
2. Combinatorial Optimisation
   - Constraint Problems
   - Solving Technologies
   - Modelling
3. Course AD3
   - Contents
   - New Concepts
   - Learning Outcomes
   - Organisation
## Example (The touristic town competition, TTC)

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Birka</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Falun</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Lund</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mora</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Sigtuna</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>Uppsala</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>Västerås</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
</tbody>
</table>

**Constraints to be satisfied:**

- Equal jury size: Every town is evaluated by 3 judges.
- Equal travel load: Every judge evaluates 3 towns.
- Fairness: Every town pair has 1 judge in common.
Example (The touristic town competition, TTC)

<table>
<thead>
<tr>
<th></th>
<th>Alva</th>
<th>Dan</th>
<th>Eva</th>
<th>Jim</th>
<th>Leo</th>
<th>Mia</th>
<th>Ulla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Birka</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Falun</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Lund</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Mora</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Sigtuna</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Uppsala</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Västerås</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Constraints to be satisfied:

- Equal jury size: Every town is evaluated by 3 judges.
- Equal travel load: Every judge evaluates 3 towns.
- Fairness: Every town pair has 1 judge in common.
Example (TTC integer model: ✓ ⇐ 1 and – ⇐ 0)

1 \text{int: nbrTowns; int: nbrJudges;}
2 \text{set of int: Towns = 1..nbrTowns;}
3 \text{set of int: Judges = 1..nbrJudges;}
4 \text{int: jurySize; int: travLoad; int: fairness;}
5 \text{array[Towns,Judges] of var 0..1: TTC;}
6 \text{solve satisfy;}
7 \text{constraint forall(t in Towns)}
8 \hspace{1em} (jurySize = \text{sum(TTC[t,..]));}
9 \text{constraint forall(j in Judges)}
10 \hspace{1em} (travLoad = \text{sum(TTC[..,j]));}
11 \text{constraint forall(t1, t2 in Towns where t1 < t2)}
12 \hspace{1em} (fairness = \text{sum(j in Judges) (TTC[t1,j] \times TTC[t2,j]));}

Example (Instance data for the Sweden TTC instance)

1 \text{nbrTowns = 7; nbrJudges = 7;}
2 \text{jurySize = 3; travLoad = 3; fairness = 1;
### Example (Idea for another TTC model)

<table>
<thead>
<tr>
<th>Town</th>
<th>Judges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Birka</td>
<td>{Alva, Dan, Eva}</td>
</tr>
<tr>
<td>Falun</td>
<td>{Alva, Jim, Leo}</td>
</tr>
<tr>
<td>Lund</td>
<td>{Alva, Mia, Ulla}</td>
</tr>
<tr>
<td>Mora</td>
<td>{Dan, Jim, Mia}</td>
</tr>
<tr>
<td>Sigtuna</td>
<td>{Dan, Leo, Ulla}</td>
</tr>
<tr>
<td>Uppsala</td>
<td>{Eva, Jim, Ulla}</td>
</tr>
<tr>
<td>Västerås</td>
<td>{Eva, Leo, Mia}</td>
</tr>
</tbody>
</table>

**Constraints to be satisfied:**

- Equal jury size: Every town is evaluated by 3 judges.
- Equal travel load: Every judge evaluates 3 towns.
- Fairness: Every town pair has 1 judge in common.
Example (TTC set model: each town has a judge set)

array[Towns] of var set of Judges: TTC;

(jurySize = card(TTC[t]));

(travLoad = sum(t in Towns)(bool2int(j in TTC[t])));

(fairness = card(TTC[t1] inter TTC[t2]));

Example (Instance data for the Sweden TTC instance)

nbrTowns = 7; nbrJudges = 7;
jurySize = 3; travLoad = 3; fairness = 1;
Example (Doctor rostering)

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

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

**Objective function to be minimised:**
- Cost: \ldots
Example (Doctor rostering)

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

**Constraints to be satisfied:**

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

**Objective function to be minimised:**
- Cost: ...
Example (Doctor rostering)

```plaintext
set of int: Days = 1..7;
set of int: Mon2Fri = 1..5;
set of int: Doctors = 1..5;
enum: ShiftTypes = {app, call, oper, none};
array[Doctors,Days] of var ShiftTypes: Roster;
solve minimize ...; % objective function
constraint forall(d in Days)
  (count(Roster[..,d],call) = 1);
constraint forall(w in Mon2Fri)
  (count(Roster[..,w],oper) <= 2);
constraint count(Roster,oper) >= 7;
constraint count(Roster,app) >= 4;
constraint forall(d in Doctors)
  (regular(Roster[d,..], (oper none|app|call|none)*));
... % other constraints
```
Example (Sudoku model)

```plaintext
array[1..9,1..9] of var 1..9: Sudoku;
... % load the hints
solve satisfy;
constraint forall(r in 1..9) (alldifferent(Sudoku[r,..]));
constraint forall(c in 1..9) (alldifferent(Sudoku[..,c]));
constraint forall(i,j in {1,4,7}) (alldifferent(Sudoku[i..i+2,j..j+2]));
```
Example (Sudoku model)

1. `array[1..9,1..9] of var 1..9: Sudoku;`
2. `... % load the hints`
3. `solve satisfy;`
4. `constraint forall(r in 1..9) (alldifferent(Sudoku[r, ..]));`
5. `constraint forall(c in 1..9) (alldifferent(Sudoku[.., c]));`
6. `constraint forall(i, j in {1, 4, 7}) (alldifferent(Sudoku[i..i+2, j..j+2]));`
Modelling Languages

The following fully declarative modelling languages are powerful enough to encode NP-hard problems:

- **Boolean satisfiability solving (SAT):** satisfy a set of disjunctions of possibly negated Boolean variables.
Modelling Languages

The following fully declarative modelling languages are powerful enough to encode NP-hard problems:

- **Boolean satisfiability solving (SAT):** *satisfy* a set of disjunctions of possibly negated Boolean variables.
- **Mixed integer programming (MIP):** *satisfy* a set of linear equalities (=) and inequalities (<, ≤, ≥, >), but not disequalities (≠), over real-number variables and integer variables weighted by real-number constants, such that a linear objective function is *optimised*.
Modelling Languages

The following fully declarative modelling languages are powerful enough to encode NP-hard problems:

- **Boolean satisfiability solving (SAT):** satisfy a set of disjunctions of possibly negated Boolean variables.

- **Mixed integer programming (MIP):** satisfy a set of linear equalities (=) and inequalities (<, ≤, ≥, >), but not disequalities (≠), over real-number variables and integer variables weighted by real-number constants, such that a linear objective function is optimised.

- **SAT modulo theories (SMT), answer-set programming (ASP), and constraint programming (CP) without or with lazy clause generation (LCG) do not have such small standardised low-level modelling languages, but enable the higher level of the previous sample models.**
Modelling Languages

The following fully declarative modelling languages are powerful enough to encode NP-hard problems:

- **Boolean satisfiability solving (SAT):** satisfy a set of disjunctions of possibly negated Boolean variables.
- **Mixed integer programming (MIP):** satisfy a set of linear equalities (\(=\)) and inequalities (\(<, \leq, \geq, >\)), but not disequalities (\(\neq\)), over real-number variables and integer variables weighted by real-number constants, such that a linear objective function is optimised.
- **SAT modulo theories (SMT), answer-set programming (ASP), and constraint programming (CP) without or with lazy clause generation (LCG) do not have such small standardised low-level modelling languages, but enable the higher level of the previous sample models.**
  
  In course 1DL449: Modelling, we use such higher-level models to drive CP, MIP, SAT, SMT, ... solvers.
Outline

1. The End of Course AD2

2. Combinatorial Optimisation
   - Constraint Problems
   - Solving Technologies
   - Modelling
   - Solvers

3. Course AD3
   - Contents
   - New Concepts
   - Learning Outcomes
   - Organisation
Combinatorial Solving Technologies

With general-purpose solvers, taking a model as input:
- Boolean satisfiability (SAT)
- SAT modulo theories (SMT)
- (Mixed) integer programming (IP and MIP)
- Constraint programming (CP)
- CP with lazy clause generation (LCG)
- Answer-set programming (ASP)
- . . .

Methodologies, *usually* without modelling languages:
- Dynamic programming (DP)
- Greedy algorithms
- Stochastic local search (SLS)
- Genetic algorithms (GA)
- Ant colony optimisation (ACO)
- Approximation algorithms
- . . .
Combinatorial Solving Technologies

With general-purpose solvers, taking a model as input:
- Boolean satisfiability (SAT) in AD3
- SAT modulo theories (SMT) in AD3
- (Mixed) integer programming (IP and MIP) in AD3
- Constraint programming (CP) in 1DL441: COCP
- CP with lazy clause generation (LCG)
- Answer-set programming (ASP)
- . . .

Methodologies, usually without modelling languages:
- Dynamic programming (DP) in 1DL231: AD2
- Greedy algorithms in 1DL231: AD2
- Stochastic local search (SLS) in AD3
- Genetic algorithms (GA)
- Ant colony optimisation (ACO)
- Approximation algorithms in AD3
- . . .
Solvers:

- **Black-box solvers** (for SAT, SMT, MIP, ASP, . . .) have general-purpose **search + inference + relaxation** that **cannot** be influenced by the modeller.
Solvers:

- **Black-box solvers** (for SAT, SMT, MIP, ASP, ... ) have general-purpose **search + inference + relaxation** that **cannot** be influenced by the modeller. 📜 AD3

- **White-box solvers** (for CP, ...) have general-purpose **search + inference + relaxation** that **can** be influenced, **if** desired, by the modeller. 📜 1DL441: COCP
Solvers:

- **Black-box solvers** (for SAT, SMT, MIP, ASP, ...) have general-purpose **search + inference + relaxation** that **cannot** be influenced by the modeller.  

- **White-box solvers** (for CP, ...) have general-purpose **search + inference + relaxation** that **can** be influenced, if desired, by the modeller.  

- **Special-purpose solvers** (for TSP, ...) exist for pure problems, where there are **no** side constraints.
Solvers:

- **Black-box solvers** (for SAT, SMT, MIP, ASP, ...) have general-purpose search + inference + relaxation that **cannot** be influenced by the modeller.  

- **White-box solvers** (for CP, ...) have general-purpose search + inference + relaxation that **can** be influenced, if desired, by the modeller.  

- **Special-purpose solvers** (for TSP, ...) exist for pure problems, where there are **no** side constraints.

**Modelling is an Art:**
Solvers:

- **Black-box solvers** (for SAT, SMT, MIP, ASP, ...) have general-purpose search + inference + relaxation that **cannot** be influenced by the modeller. 

- **White-box solvers** (for CP, ...) have general-purpose search + inference + relaxation that **can** be influenced, if desired, by the modeller. 

- **Special-purpose solvers** (for TSP, ...) exist for pure problems, where there are **no** side constraints.

**Modelling is an Art:**

- Different models of a problem may take different time on the same solver for the same instance.
Solvers:

- **Black-box solvers** (for SAT, SMT, MIP, ASP, . . . ) have general-purpose search + inference + relaxation that cannot be influenced by the modeller.  

- **White-box solvers** (for CP, . . . ) have general-purpose search + inference + relaxation that can be influenced, if desired, by the modeller.  

- **Special-purpose solvers** (for TSP, . . . ) exist for pure problems, where there are no side constraints.

**Modelling is an Art:**

- 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.
Solvers:

- **Black-box solvers** (for SAT, SMT, MIP, ASP, ...) have general-purpose search + inference + relaxation that cannot be influenced by the modeller. 

- **White-box solvers** (for CP, ...) have general-purpose search + inference + relaxation that can be influenced, if desired, by the modeller.

- **Special-purpose solvers** (for TSP, ...) exist for pure problems, where there are no side constraints.

**Modelling is an Art:**

- 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 or technologies may take different time for the same instance on the same model.
Solvers:

- **Black-box solvers** (for SAT, SMT, MIP, ASP, ...) have general-purpose search + inference + relaxation that cannot be influenced by the modeller.

- **White-box solvers** (for CP, ...) have general-purpose search + inference + relaxation that can be influenced, if desired, by the modeller.

- **Special-purpose solvers** (for TSP, ...) exist for pure problems, where there are no side constraints.

**Modelling is an Art:**

- 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 or technologies may take different time for the same instance on the same model.

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

1. The End of Course AD2

2. Combinatorial Optimisation
   - Constraint Problems
   - Solving Technologies
   - Modelling
   - Solvers

3. Course AD3
   - Contents
   - New Concepts
   - Learning Outcomes
   - Organisation
Outline

1. The End of Course AD2
2. Combinatorial Optimisation
   - Constraint Problems
   - Solving Technologies
   - Modelling
   - Solvers
3. Course AD3
   - Contents
   - New Concepts
   - Learning Outcomes
   - Organisation
1. Stochastic local search (SLS)
2. Mixed integer programming (MIP)
3. Amortised analysis (CLRS3: Chapter 17)
4. Probabilistic analysis (Chapter 5)
5. Randomised algorithms: universal hashing, ... (Section 11.3.3)
6. Proving NP-completeness by reduction (Chapter 34)
7. Boolean satisfiability (SAT)
8. SAT modulo theories (SMT)
9. Approximation algorithms (Chapter 35)

CLRS3 Textbook:
Introduction to Algorithms (3rd edition) (errata).
Outline

1. The End of Course AD2
2. Combinatorial Optimisation
   - Constraint Problems
   - Solving Technologies
   - Modelling
   - Solvers
3. Course AD3
   - Contents
   - New Concepts
   - Learning Outcomes
   - Organisation
In a **probabilistic algorithm analysis**, we use probability theory: knowing or assuming the distribution of the inputs, we compute the **average-case time** (as opposed to the worst-case time) of a deterministic algorithm.
In a **probabilistic algorithm analysis**, we use probability theory: knowing or assuming the distribution of the inputs, we compute the **average-case time** (as opposed to the worst-case time) of a deterministic algorithm.

**Example**

The brute-force string matching algorithm for finding all occurrences of a pattern $P$ of length $m$ within a text $T$ of length $n \geq m$ takes $O(n - m + 1)$ time on average when $P$ and $T$ are random strings, but this is a completely unreasonable assumption. (Chapter 32 in CLRS3)
In a **probabilistic algorithm analysis**, we use probability theory: knowing or assuming the distribution of the inputs, we compute the **average-case time** (as opposed to the worst-case time) of a deterministic algorithm.

**Example**

The brute-force string matching algorithm for finding all occurrences of a pattern $P$ of length $m$ within a text $T$ of length $n \geq m$ takes $O(n - m + 1)$ time on average when $P$ and $T$ are random strings, but this is a completely unreasonable assumption. (Chapter 32 in CLRS3)

Probabilistic analysis helps gain insight into a problem and helps design an efficient algorithm for it, when we have a reasonable assumption on the distribution of the inputs.
A randomised algorithm (as opposed to a deterministic algorithm) itself makes random choices, independently of the actual distribution of the inputs. We refer to the time of a randomised algorithm as expected time (not average time).
A randomised algorithm (as opposed to a deterministic algorithm) itself makes random choices, independently of the actual distribution of the inputs. We refer to the time of a randomised algorithm as expected time (not average time).

**Examples**

- A randomised algorithm by Karger-Klein-Tarjan (1993) computes in $O(V + E)$ expected time a minimum spanning tree (MST) of a connected undirected graph with vertex set $V$ and edge set $E$. (Chapter 23)
A randomised algorithm (as opposed to a deterministic algorithm) itself makes random choices, independently of the actual distribution of the inputs. We refer to the time of a randomised algorithm as expected time (not average time).

Examples

- A randomised algorithm by Karger-Klein-Tarjan (1993) computes in $O(V + E)$ expected time a minimum spanning tree (MST) of a connected undirected graph with vertex set $V$ and edge set $E$. (Chapter 23)

- A randomised algorithm computes in $O(m)$ expected time a prime number larger than $m$ for fingerprinting in the Rabin-Karp string matcher. (Chapter 32)
A randomised algorithm (as opposed to a deterministic algorithm) itself makes random choices, independently of the actual distribution of the inputs. We refer to the time of a randomised algorithm as expected time (not average time).

Examples

- A randomised algorithm by Karger-Klein-Tarjan (1993) computes in $\mathcal{O}(V + E)$ expected time a minimum spanning tree (MST) of a connected undirected graph with vertex set $V$ and edge set $E$. (Chapter 23)

- A randomised algorithm computes in $\mathcal{O}(m)$ expected time a prime number larger than $m$ for fingerprinting in the Rabin-Karp string matcher. (Chapter 32)

Many randomised algorithms have no worst-case input!
In an **amortised analysis**, we compute the worst-case time of a *chain* of data-structure operations, and average it over the operations. We refer to this time as an **amortised time** (as opposed to an average-case time, as no probability is used here, and the possibly non-tight worst-case time).
In an amortised analysis, we compute the worst-case time of a chain of data-structure operations, and average it over the operations. We refer to this time as an amortised time (as opposed to an average-case time, as no probability is used here, and the possibly non-tight worst-case time).

**Examples**

- A chain of \( m \) find-and-compress-paths or union-by-rank operations on disjoint sets of \( n \) items takes \( O(\alpha(n) \cdot m) \) time, where \( \alpha(n) \leq 4 \) in practice. (Chapter 21)
In an **amortised analysis**, we compute the worst-case time of a *chain* of data-structure operations, and average it over the operations. We refer to this time as an **amortised time** (as opposed to an average-case time, as no probability is used here, and the possibly non-tight worst-case time).

**Examples**

- A chain of *m* find-and-compress-paths or union-by-rank operations on disjoint sets of *n* items takes \( O(\alpha(n) \cdot m) \) time, where \( \alpha(n) \leq 4 \) in practice. (Chapter 21)

- In a Fibonacci heap of *n* items, extracting a minimum takes \( O(\lg n) \) amortised time, and decreasing a key takes \( O(1) \) amortised time. (Chapter 19; not in AD2)
In an amortised analysis, we compute the worst-case time of a chain of data-structure operations, and average it over the operations. We refer to this time as an amortised time (as opposed to an average-case time, as no probability is used here, and the possibly non-tight worst-case time).

Examples
- A chain of \( m \) find-and-compress-paths or union-by-rank operations on disjoint sets of \( n \) items takes \( O(\alpha(n) \cdot m) \) time, where \( \alpha(n) \leq 4 \) in practice.  
  (Chapter 21)
- In a Fibonacci heap of \( n \) items, extracting a minimum takes \( O(\lg n) \) amortised time, and decreasing a key takes \( O(1) \) amortised time.  
  (Chapter 19; not in AD2)
- Prim’s MST algorithm takes at worst \( O(E + V \lg V) \) time when using a Fibonacci heap.  
  (Chapter 23)
Dealing in polynomial time with (instances of) optimisation problems where brute-force or exact solving is too costly:

- A **greedy algorithm** builds a feasible solution variable by variable, making locally optimal choices in the hope of reaching an optimal solution. Greedy algorithms build either **provably optimal** solutions (for example, Prim’s MST algorithm and Dijkstra’s single-source shortest paths algorithm) or **at-best optimal** solutions.
Dealing in polynomial time with (instances of) optimisation problems where brute-force or exact solving is too costly:

- A **greedy algorithm** builds a feasible solution variable by variable, making locally optimal choices in the hope of reaching an optimal solution. Greedy algorithms build either **provably optimal** solutions (for example, Prim’s MST algorithm and Dijkstra’s single-source shortest paths algorithm) or **at-best optimal** solutions.

- A **local search algorithm** repairs a possibly infeasible candidate solution, by reassigning some variables at every iteration, until an allocated resource (such as an iteration count or a time budget) is exhausted, in the hope of reaching a feasible or even optimal solution.
Dealing in polynomial time with (instances of) optimisation problems where brute-force or exact solving is too costly:

- A **greedy algorithm** builds a feasible solution variable by variable, making locally optimal choices in the hope of reaching an optimal solution. Greedy algorithms build either **provably optimal** solutions (for example, Prim’s MST algorithm and Dijkstra’s single-source shortest paths algorithm) or **at-best optimal** solutions.

- A **local search algorithm** repairs a possibly infeasible candidate solution, by reassigning some variables at every iteration, until an allocated resource (such as an iteration count or a time budget) is exhausted, in the hope of reaching a feasible or even optimal solution.

- An **approximation algorithm** for an NP-hard problem builds a feasible solution whose objective value is **provably** within a known factor of the optimum.
Dealing in polynomial time with (instances of) optimisation problems where brute-force or exact solving is too costly:

- A **greedy algorithm** builds a feasible solution variable by variable, making locally optimal choices in the hope of reaching an optimal solution. Greedy algorithms build either **provably optimal** solutions (for example, Prim’s MST algorithm and Dijkstra’s single-source shortest paths algorithm) or **at-best optimal** solutions.

- A **local search algorithm** repairs a possibly infeasible candidate solution, by reassigning some variables at every iteration, until an allocated resource (such as an iteration count or a time budget) is exhausted, in the hope of reaching a feasible or even optimal solution.

- An **approximation algorithm** for an NP-hard problem builds a feasible solution whose objective value is **provably** within a known factor of the optimum.

All techniques are orthogonal: there exist randomised local search algorithms, greedy approximation algorithms, etc.
Outline

1. The End of Course AD2
2. Combinatorial Optimisation
   - Constraint Problems
   - Solving Technologies
   - Modelling
   - Solvers
3. Course AD3
   - Contents
   - New Concepts
   - Learning Outcomes
   - Organisation
In order to pass, the student must be able to:

- analyse NP-completeness of an algorithmic problem;
- use advanced algorithm analysis methods, such as amortised analysis and probabilistic analysis;
- use advanced algorithm design methods in order to approach hard algorithmic problems in a pragmatic way, such as by using:
  - randomised algorithms: universal hashing, …
  - approximation algorithms
  - local search: simulated annealing, tabu search, …
  - mixed integer programming (MIP)
  - Boolean satisfiability (SAT)
  - SAT modulo theories (SMT)
Outline

1. The End of Course AD2
2. Combinatorial Optimisation
   - Constraint Problems
   - Solving Technologies
   - Modelling
   - Solvers
3. Course AD3
   - Contents
   - New Concepts
   - Learning Outcomes
   - Organisation
Period 3: January to March: your budget = 133.3 hours

11 lectures, taught in English

2 assignments with 2 help sessions, 1 grading session, and 1 solution session per assignment, on 2 problems each, to be done in pairs upon large class: budget = 28 hours / assignment / student (2 credits)

1 oral presentation, done in pairs upon large class: budget = 15 hours / student, and
1 written closed-book exam of 3 hours, to be done alone, even if the class is large (3 credits)

Prerequisites: Algorithms and Data Structures 2 (course 1DL231) or equivalent