Topic 6: Case Studies
(Version of 3rd August 2023)

Pierre Flener and Gustav Björdal

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

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

Outline

1. Black-Hole Patience
2. Cost-Aware Scheduling
3. Warehouse Location
4. Sport Scheduling

Outline

Move all the cards into the black hole. A fan top card can be moved if it is one rank apart from the black-hole top card, independently of suit (♠, ♣, ♦, ♥); aces (A,1) and kings (K,13) are a rank apart.

Card encoding: ♠: 1..13, ♣: 14..26, ♦: 27..39, ♥: 40..52.

The cards c_1 and c_2 are one rank apart if and only if

$$(c_1 \bmod 13) - (c_2 \bmod 13) \in \{-12, -1, 1, 12\}$$

Define a predicate and avoid `mod` on decision variables, by precomputation:

```
predicate rankApart(var 1..52: c1, var 1..52: c2) =
  let { array[1..52] of int: Rank = [i mod 13 | i in 1..52] }
    in Rank[c1] - Rank[c2] in {-12,-1,1,12};
```

Avoid implicit `element` constraints, for better inference:

```
  table([c1,c2], [|1,2|1,13|...|1,52|2,3|...|52,40|52,51|]);
```

Move all the cards into the black hole. A fan top card can be moved if it is one rank apart from the black-hole top card, independently of suit (♠, ♣, ♦, ♥); aces (A,1) and kings (K,13) are a rank apart.

Let `Card[p]` denote the card at position `p` in the black hole.
Adjacent black-hole cards are a rank apart:

```
constraint Card[1] = 1; % the card at position 1 is A♠
constraint forall(p in 1..51)(rankApart(Card[p],Card[p+1]));
```

The black-hole cards respect the order in the given fans:

```
constraint forall(f in Fan)
  (let { var 2..52: p1; var 2..52: p2; var 2..52: p3 } in
  Card[p1]=f.top/\Card[p2]=f.mid/\Card[p3]=f.bot/\p1<p2/\p2<p3);
```

or, equivalently, but better because without the implicit `element` constraints:

```
constraint all_different(Card) /\ forall(f in Fan)
  (value_precede_chain([f.top,f.mid,f.bot],Card));
```

Let `Pos[c]` denote the position of card `c` in the black hole.
The black-hole cards respect the order in the given fans:

```
constraint Pos[1] = 1; % the position of card A♠ is 1
constraint forall(f in Fan)
  (Pos[f.top] < Pos[f.mid] /\ Pos[f.mid] < Pos[f.bot]);
```

How to model “adjacent black-hole cards are a rank apart” with the `Pos[c]`?!

Let us use the `Pos[c]` for the second constraint, as mutually redundant with the `Card[p]` for the first constraint, and 2-way channel between them.

Observe that $\forall$`c`, `p` $\in 1..52$: `Card[p] = c` $\Leftrightarrow$ `Pos[c] = p`.
Seen as functions, `Card` and `Pos` are each other’s inverse:

```
constraint inverse(Card,Pos) :: domain_propagation; % Topic 8
```

This model ↗ + ↗ with mutually redundant decision variables and the 2-way channelling constraint is much faster (at least on a CP or LCG solver) than the model on the previous slide with only the `Card` decision variables.

Outline

Energy-Cost-Aware Scheduling

Consider the core of CSPlib problem 059. Given are:

- Machines, each machine having several capacitated reusable resources.
- Jobs, each job having a duration, earliest start time, latest end time, a consumption of energy (which is an overall consumable resource, not a reusable resource of the machines), and requirements for the reusable resources of the machines.
- A time horizon, each time step having a predicted energy cost.

Schedule the jobs and allocate them to machines, so that:

1. No job starts too early or ends too late.
2. No resource capacity of any machine is ever exceeded.
3. The total energy cost is minimal.

We show that precomputing a 2d array with the energy cost of each job for each possible start time boosts everything.

Parameters

```
enum Resources; % say: {cpu,ram,io};
int: nMachines; set of int: Machines = 1..nMachines;
array[Machines,Resources] of int: Capacity;
int: nTimeSteps; % say: 288, for every 5 minutes over 24h
set of int: Times = 0..nTimeSteps; % time points
set of int: Steps = 1..nTimeSteps; % time step s is from point s-1 to point s
array[Steps] of float: EnergyCost; % EnergyCost[s] €/kWh during time step s
int: nJobs; set of int: Jobs = 1..nJobs;
array[Jobs] of Steps: Duration;  % job j lasts Duration[j] steps
array[Jobs] of Times: EarliestS; % job j starts >= EarliestS[j]
array[Jobs] of Times: LatestEnd; % job j ends   <= LatestEnd[j]
array[Jobs] of int: Energy;      % job j consumes Energy[j] kWh
array[Jobs,Resources] of int: Requirement;
```

In the instance `sample03` of CSPlib problem 059 we have:

```
EnergyCost[119..128] = [0.04732, 0.04732, 0.08093, 0.08093, 0.08093, 0.08093,
  0.08093, 0.08093, 0.08619, 0.08619]
```

A job of 6 steps & 1151 kWh costs $\lfloor 1151 \cdot (0.04732 \cdot 2 + 0.08093 \cdot 4) \rfloor = 481$€ at time 118, and $\lfloor 1151 \cdot (0.08093 \cdot 4 + 0.08619 \cdot 2) \rfloor = 571$€ at time 122.

Model


```
array[Jobs] of var Times: Start; % job j starts at time Start[j]
array[Jobs] of var Machines: Machine; % job j runs on Machine[j]
% (1) No job starts too early or ends too late:
constraint forall(j in Jobs)
   (Start[j] in EarliestS[j]..LatestEnd[j]-Duration[j]);
% (2) No resource capacity of any machine is ever exceeded:
constraint forall(m in Machines, r in Resources)(cumulative(Start, Duration,
   [(Machine[j] = m) * Requirement[j,r] | j in Jobs], Capacity[m,r]));
... % constraints for the rest of the problem
array[Jobs] of var 0..floor(max(Energy)*sum(EnergyCost)): Cost;% j is Cost[j]€
... % see the next slide!
solve minimize sum(Cost) + ...;
```

Define the decision variables `Cost[j]` without precomputation:

```
constraint forall(j in Jobs)(Cost[j] = sum(s in Steps)(if Start[j]+1 <= s /\
  s <= Start[j]+Duration[j] then floor(Energy[j]*EnergyCost[s]) else 0 endif));
```

For `sample03`, with 100 jobs and 288 time steps, this compiles under Gecode in over 20 seconds into 12 MB of FlatZinc code, with 74,137 constraints and 66,828 decision variables, due to the use of `if` θ `then` ϕ `else` ψ `endif` with a test θ that depends on decision variables (the `Start[j]` here).

Define the decision variables `Cost[j]` with precomputation of an array of derived parameters:

```
% JobCost[j,t] = energy cost of job j if j starts at time t (with dummy values
  if t+Duration[j] > nTimeSteps):
array[Jobs,Times] of int: JobCost = array2d(Jobs, Times,
  [floor(Energy[j] * sum(EnergyCost[t+1..min(t+Duration[j],nTimeSteps)]))
  | j in Jobs, t in Times]); % round the sum, not its terms!
constraint forall(j in Jobs)(Cost[j] = JobCost[j,Start[j]]);
```

For `sample03`, this model compiles very fast under Gecode into only 343 KB of FlatZinc code, with 100 constraints and 100 decision variables, and a feasible solution is found six times faster.

Outline

The Warehouse Location Problem (WLP)

A company considers opening warehouses at some candidate locations in order to supply its existing shops:

- Each candidate warehouse has the same maintenance cost.
- Each candidate warehouse has a supply capacity, which is the maximum number of shops it can supply.
- The supply cost to a shop depends on the supplying warehouse.

Determine which candidate warehouses actually to open, and which of them supplies which shops, so that:

1. Each shop is supplied by exactly one actually opened warehouse.
2. Each actually opened warehouse supplies a number of shops that is at most equal to its supply capacity.
3. The sum of the actually incurred maintenance costs and supply costs is minimal.

WLP: Sample Instance Data

$$\texttt{Shops} = \{\text{Shop}_1, \text{Shop}_2, \ldots, \text{Shop}_{10}\}$$

$$\texttt{Warehouses} = \{\text{Berlin, London, Ankara, Paris, Rome}\}$$

$$\texttt{maintCost} = 30$$

Capacity =

Berlin	London	Ankara	Paris	Rome
1	4	2	1	3

SupplyCost =

	Berlin	London	Ankara	Paris	Rome
Shop_1	20	24	11	25	30
Shop_2	28	27	82	83	74
Shop_3	74	97	71	96	70
Shop_4	2	55	73	69	61
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Shop_{10}	47	65	55	71	95

UPPSALA UNIVERSITET

WLP Model 1: Decision Variables

Automatic enforcement of the total-function constraint (1):

$$\texttt{Supplier} = \begin{array}{cccc} \text{Shop}_1 & \text{Shop}_2 & \cdots & \text{Shop}_{10} \\ \hline \multicolumn{1}{|c|}{\in \texttt{Warehouses}} & \multicolumn{1}{c|}{\in \texttt{Warehouses}} & \multicolumn{1}{c|}{\cdots} & \multicolumn{1}{c|}{\in \texttt{Warehouses}} \\ \hline \end{array}$$

`Supplier[s]` denotes the supplier warehouse for shop `s`.

Variables redundant with `Supplier`, but not mutually, as less informative:

$$\texttt{Open} = \begin{array}{ccccc} \text{Berlin} & \text{London} & \text{Ankara} & \text{Paris} & \text{Rome} \\ \hline \multicolumn{1}{|c|}{\in \texttt{0..1}} & \multicolumn{1}{c|}{\in \texttt{0..1}} & \multicolumn{1}{c|}{\in \texttt{0..1}} & \multicolumn{1}{c|}{\in \texttt{0..1}} & \multicolumn{1}{c|}{\in \texttt{0..1}} \\ \hline \end{array}$$

`Open[w] = 1` if and only if warehouse `w` is actually opened.

☞ Our chosen array names always reflect total functions.

WLP Model 1: Objective

```
solve minimize maintCost * sum(Open)
  + sum(s in Shops)(SupplyCost[s,Supplier[s]]);
```

The first term is the total maintenance cost, expressed as the product of the warehouse maintenance cost by the number of actually opened warehouses.

The second term is the total supply cost, expressed as the sum over all shops of their actually incurred supply costs.

Notice the implicit use of the `element` predicate,
as the column index `Supplier[s]` to `SupplyCost` is a decision variable.

If warehouse `w` has maintenance cost `MaintCost[w]`, then the first term becomes `sum(w in Warehouses)(MaintCost[w] * Open[w])`.

WLP Model 1: Channelling Constraint

One-way channelling constraint from the `Supplier[s]` decision variables to some of their redundant `Open[w]` decision variables (as not all `Open[w]` are fixed this way):

```
constraint forall(s in Shops)(Open[Supplier[s]] = 1);
```

The supplier warehouse of each shop is actually opened.

Notice the implicit use of the `element` predicate, as the index `Supplier[s]` to `Open` is a decision variable.

How do the remaining `Open[w]` become `0`? Upon minimisation!

WLP Model 1: Channelling Constraint

Alternative: One-way channelling constraint from the `Supplier[s]` decision variables to all of their redundant `Open[w]` decision variables, but not vice-versa:

```
constraint forall(w in Warehouses)
   (Open[w] = (exists(s in Shops)(Supplier[s]=w)));
```

A warehouse is opened if and only if there exists a shop that it supplies.

Make experiments to find out which channelling is better.
We will revisit this issue in Topic 8: Inference & Search in CP & LCG, and in Topic 9: Modelling for CBLS.

Nothing changes if `Open` is an array of Boolean decision variables (instead of integer decision variables).

WLP Model 1: Capacity Constraint

Capacity constraint (2), using a version of `global_cardinality` with given lower and upper bounds rather than decision variables for the counts:

```
constraint global_cardinality_closed
  (Supplier, Warehouses, [0 | w in Warehouses], Capacity);
```

Each actually opened warehouse is a supplier of a number of shops that is at most equal to its supply capacity.

Which symmetries are there?

- There are no problem symmetries.
- We introduced no symmetries into the model.
- There may be instance symmetries: indistinguishable shops, or indistinguishable warehouses, or both.

WLP Model 2

Drop the array `Open` of redundant decision variables as well as its channelling constraint, and reformulate the first term of the objective function as follows:

```
maintCost *
sum(w in Warehouses)(exists(s in Shops)(Supplier[s]=w))
```

We can alternatively use the `nvalue` constrained function:

```
maintCost *
nvalue(Supplier)
```

This alternative formulation cannot be generalised for warehouse-specific maintenance costs.

For a speed comparison, see Topic 8: Inference & Search in CP & LCG.

Redundancy elimination may pay off, but it may just as well be the converse.

But this is hard to guess, as human intuition may be weak.

WLP Model 3: Decision Variables

No automatic enforcement of the total-function constraint (1):

$$\texttt{Supply} = \begin{array}{c|c|c|c|c|c|} & \text{Berlin} & \text{London} & \text{Ankara} & \text{Paris} & \text{Rome} \\ \hline \text{Shop}_1 & \in \texttt{0..1} & \in \texttt{0..1} & \in \texttt{0..1} & \in \texttt{0..1} & \in \texttt{0..1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \text{Shop}_{10} & \in \texttt{0..1} & \in \texttt{0..1} & \in \texttt{0..1} & \in \texttt{0..1} & \in \texttt{0..1} \\ \hline \end{array}$$

`Supply[s,w] = 1` if and only if shop `s` is supplied by warehouse `w`.

Redundant decision variables (as in Model 1):

$$\texttt{Open} = \begin{array}{|c|c|c|c|c|} \text{Berlin} & \text{London} & \text{Ankara} & \text{Paris} & \text{Rome} \\ \hline \in \texttt{0..1} & \in \texttt{0..1} & \in \texttt{0..1} & \in \texttt{0..1} & \in \texttt{0..1} \\ \hline \end{array}$$

`Open[w] = 1` if and only if warehouse `w` is actually opened.

WLP Model 3: Objective

The objective can now be expressed in linear fashion:

```
solve minimize maintCost * sum(Open)
  + sum(s in Shops, w in Warehouses)
    (SupplyCost[s,w] * Supply[s,w]);
```

The first term is the total maintenance cost, expressed (as in Model 1) as the product of the warehouse maintenance cost by the number of actually opened warehouses.

The second term is the total supply cost, expressed as the sum over all shops and warehouses of their actually incurred supply costs: each decision variable `Supply[s,w]` is weighted by the parameter `SupplyCost[s,w]`.

WLP Model 3: Constraints

The total-function constraint (1) now needs to be modelled, and can be expressed in linear fashion (that is, without using `count`):

```
constraint forall(s in Shops)(sum(Supply[s,..]) = 1);
```

Each shop is supplied by exactly one actually opened warehouse.

WLP Model 3: Constraints (end)

Capacity constraint (2), in isolation:

```
constraint forall(w in Warehouses)
  (sum(Supply[..,w]) <= Capacity[w]);
```

One-way channelling constraint, in isolation:

```
constraint forall(w in Warehouses)
  (sum(Supply[..,w]) > 0  <->  Open[w] = 1);
```

or, one-way channelling without reification, upon exploiting minimisation:

```
constraint forall(w in Warehouses)
  (forall(s in Shops)(Supply[s,w] <= Open[w]));
```

Capacity (2) and second one-way channelling constraints combined:

```
constraint forall(w in Warehouses)
  (sum(Supply[..,w]) <= Capacity[w] * Open[w]);
```

All constraints are linear (in)equalities: this is an IP model!

Outline

1. Black-Hole Patience
2. Cost-Aware Scheduling
3. Warehouse Location
4. **Sport Scheduling**

UPPSALA UNIVERSITET

Black-Hole Patience

Cost-Aware Scheduling

Warehouse Location

Sport Scheduling

The Sport Scheduling Problem (SSP)

Find a schedule in $\texttt{Periods} \times \texttt{Weeks} \to \texttt{Teams} \times \texttt{Teams}$ for

- $|\texttt{Teams}| = \texttt{n}$ and $\texttt{n}$ is even (note that only $\texttt{n=4}$ is unsatisfiable)
- $|\texttt{Weeks}| = \texttt{n-1}$
- $|\texttt{Periods}| = \texttt{n}/2$ periods per week

subject to the following constraints:

1. Each possible game is played exactly once.
2. Each team plays exactly once per week.
3. Each team plays at most twice per period.

Idea for a model, and a solution for $\texttt{n=8}$:

	Wk 1	Wk 2	Wk 3	Wk 4	Wk 5	Wk 6	Wk 7
P 1	1 vs 2	1 vs 3	2 vs 6	3 vs 5	4 vs 7	4 vs 8	5 vs 8
P 2	3 vs 4	2 vs 8	1 vs 7	6 vs 7	6 vs 8	2 vs 5	1 vs 4
P 3	5 vs 6	4 vs 6	3 vs 8	1 vs 8	1 vs 5	3 vs 7	2 vs 7
P 4	7 vs 8	5 vs 7	4 vs 5	2 vs 4	2 vs 3	1 vs 6	3 vs 6

The Sport Scheduling Problem (SSP)

Find a schedule in $\mathtt{Periods} \times \mathtt{Weeks} \to \mathtt{Teams} \times \mathtt{Teams}$ for

- $|\mathtt{Teams}| = \mathtt{n}$ and $\mathtt{n}$ is even (note that only $\mathtt{n=4}$ is unsatisfiable)
- $|\mathtt{Weeks}| = \mathtt{n-1}$
- $|\mathtt{Periods}| = \mathtt{n/2}$ periods per week

subject to the following constraints:

1. Each possible game is played exactly once.
2. Each team plays exactly once per week.
3. Each team plays at most twice per period.

Idea for a model, and a solution for $\mathtt{n=8}$, with a dummy week $\mathtt{n}$ of duplicates:

	Wk 1	Wk 2	Wk 3	Wk 4	Wk 5	Wk 6	Wk 7	Wk 8
P 1	1 vs 2	1 vs 3	2 vs 6	3 vs 5	4 vs 7	4 vs 8	5 vs 8	6 vs 7
P 2	3 vs 4	2 vs 8	1 vs 7	6 vs 7	6 vs 8	2 vs 5	1 vs 4	3 vs 5
P 3	5 vs 6	4 vs 6	3 vs 8	1 vs 8	1 vs 5	3 vs 7	2 vs 7	2 vs 4
P 4	7 vs 8	5 vs 7	4 vs 5	2 vs 4	2 vs 3	1 vs 6	3 vs 6	1 vs 8

SSP Model 1: Data

Parameter:

- `int: n; constraint assert(n>=2 /\ n mod 2 =0,"Odd n");`

Useful Ranges, enumeration, and set:

- `Teams = 1..n`
- `Weeks = 1..(n-1)`
- `ExtendedWeeks = 1..n`
- `Periods = 1..(n div 2)`
- `Slots = {one, two}`
- `Games = {f * n + s | f,s in Teams where f < s}`,
  thereby breaking some symmetries, such that the game between teams `f` and `s` is uniquely identified by the natural number `f * n + s`.

Example: For `n = 4`, we get `Games = {6,7,8,11,12,16}`.

SSP Model 1: Decision Variables

Declare a 3d matrix `Team[Periods,ExtendedWeeks,Slots]` of decision variables in `Teams` (denoted `T` below), over a schedule extended by a dummy week where teams play fictitious duplicate games in the period where they would otherwise play only once, thereby strengthening constraint (3) into:

(3') Each team plays exactly twice per period.

Let `Team[p,w,s]` be the team that plays in period `p` of week `w` in game slot `s`:

$$
\texttt{Team} = \begin{array}{c|cc|cc|cc|cc|}
 & \multicolumn{2}{c}{\text{Wk } 1} & \multicolumn{2}{c}{\cdots} & \multicolumn{2}{c}{\text{Wk } n-1} & \multicolumn{2}{c}{\text{Wk } n} \\
 & \text{one} & \text{two} & \cdots & \cdots & \text{one} & \text{two} & \text{one} & \text{two} \\
\hline
\text{P } 1 & \in \texttt{T} & \in \texttt{T} & \cdots & \cdots & \in \texttt{T} & \in \texttt{T} & \in \texttt{T} & \in \texttt{T} \\
\vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\
\text{P } n/2 & \in \texttt{T} & \in \texttt{T} & \cdots & \cdots & \in \texttt{T} & \in \texttt{T} & \in \texttt{T} & \in \texttt{T} \\
\hline
\end{array}
$$

UPPSALA UNIVERSITET

SSP Model 1: Constraints

Twice-per-period constraint (3’):

```
constraint forall(p in Periods)
  (global_cardinality_closed
    (Team[p,..,..], Teams, [2 | i in 1..n]));
```

In each period, each team occurs exactly twice within the slots of the weeks. (We do not need the four-argument version of the predicate, with an array of ones as lower bounds and an array of twos as upper bounds.)

Once-per-week constraint (2):

```
constraint forall(w in ExtendedWeeks)
  (all_different(Team[..,w,..]));
```

In each week, including the dummy week, there are no duplicate teams within the slots of the periods in `Team`.

SSP Model 1: Decision Variables (revisited)

Try to state the each-game-once constraint (1) using `Team`!

Rather declare a 2d matrix `Game[Periods,Weeks]` of decision variables in `Games` over the non-extended weeks.

Let `Game[p,w]` be the game played in period `p` of week `w`:

$$\texttt{Game} = \begin{array}{r|c|c|c|} & \text{Week } 1 & \cdots & \text{Week } n-1 \\ \hline \text{Period } 1 & \in \texttt{Games} & \cdots & \in \texttt{Games} \\ \vdots & \vdots & \ddots & \vdots \\ \text{Period } n/2 & \in \texttt{Games} & \cdots & \in \texttt{Games} \\ \hline \end{array}$$

The 2d matrix `Game` is mutually redundant with the first $n-1$ 2d columns of the 3d matrix `Team`, which is over the extended weeks.

SSP Model 1: Constraints (end)

Each-game-once constraint (1):

```
constraint all_different(Game);
```

There are no duplicate game numbers in `Game`.

Two-way channelling constraint (but rather precompute and use `table`: see Topic 8: Inference & Search in CP & LCG):

```
constraint forall(p in Periods, w in Weeks)
  (Team[p,w,one] * n + Team[p,w,two] = Game[p,w]);
```

The game number in `Game` of each period and week corresponds to the teams scheduled at that time in `Team`.

The constraints (2) and (3’) are hard to formulate using `Game`.

Add the symmetry-breaking constraints of slide 29 of Topic 5: Symmetry.

SSP Model 2: Smaller Domains for `Game[p,w]` Variables

A round-robin schedule suffices to break many of the remaining symmetries:

- Restrict the games of the first week to the set
  $\{1 \text{ vs } 2\} \cup \{t+1 \text{ vs } n+2-t \mid 1 < t \leq n/2\}$
- For the remaining weeks, transform each game f vs s of the previous week into a game f' vs s', where

$$f' = \begin{cases} 1 & \text{if } f = 1 \\ 2 & \text{if } f = n \\ f+1 & \text{otherwise} \end{cases}, \text{ and } s' = \begin{cases} 2 & \text{if } s = n \\ s+1 & \text{otherwise} \end{cases}$$

The constraints (1) and (2) are now automatically enforced:
we must only find the period of each game, but not its week.

Interested in More Details?

For more details on WLP and SSP and their modelling, see:

- Van Hentenryck, Pascal.
  The OPL Optimization Programming Language.
  The MIT Press, 1999.
- Van Hentenryck, Pascal.
  Constraint and integer programming in OPL.
  INFORMS Journal on Computing, 14(4):345–372, 2002.
- Van Hentenryck, Pascal; Michel, Laurent; Perron, Laurent; and Régin, Jean-Charles.
  Constraint programming in OPL.
  PPDP 1999, pages 98–116. *Lecture Notes in Computer Science* 1702.
  Springer-Verlag, 1999.