Topic 6: Case Studies
(Version of 26th September 2018)

Pierre Flener and Gustav Björdal

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

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

Outline

UPPSALA UNIVERSITET

Outline

Black-Hole Patience

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

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

Avoiding `mod` on variables and defining a help predicate:

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

Avoiding implicit `element` constraints for better inference:

```
predicate rankApart(var 1..52: c1, var 1..52: c2) =
  table([c1,c2], [|1,2|1,13|...|1,52|2,1|...|52,40|52,51|]);
```

UPPSALA UNIVERSITET

Model

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:

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

The card order in the fans is respected in the black hole:

```
constraint forall("fan with card" c1 "on top of" c2 "on top of" c3)
  (let { var 2..52: p1; var 2..52: p2; var 2..52: p3 }
   in  Card[p1]=c1/\Card[p2]=c2/\Card[p3]=c3/\p1<p2/\p2<p3);
% constraint alldifferent(Card);  % implied by correct data!
```

or, equivalently:

```
constraint forall("fan with card" c1 "on top of" c2 "on top of" c3)
  (value_precede_chain([c1,c2,c3],Card));
% constraint alldifferent(Card);  % implied by correct data!
```

Redundant Variables and Channelling

Let `Pos[c]` denote the position of card `c` in the black hole. The card order in the fans is respected in the black hole:

```
constraint Pos[1] = 1; % the position of card A♠ is 1
constraint forall(“fan with card” c1 “on top of” c2 “on top of” c3)
  (Pos[c1] < Pos[c2] /\ Pos[c2] < Pos[c3]);
```

Let us use both the `Card` variables and the `Pos` variables, and channel between them.

Observe that $\forall \mathtt{c}, \mathtt{p} \in 1..52 : \mathtt{Card[p]=c} \Leftrightarrow \mathtt{Pos[c]=p}$. Seen as functions, `Card` and `Pos` are each other's inverse. This is the semantics of `inverse(Card,Pos)` and implies both `alldifferent(Card)` and `alldifferent(Pos)`.

This model with redundant variables and their channelling constraint is much more efficient (at least on a CP or LCG solver) than the models with only the `Card` variables.

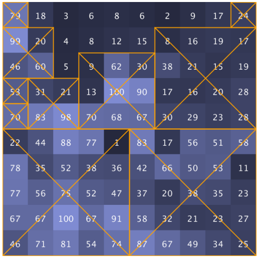
Outline

Antenna Placement Problem

Given:

- a region divided into zones, each with an expected gain if covered,
- the maintenance costs and covering areas of antenna types,
- a targeted number of antennae,

find non-overlapping antenna placements and types such that the total expected gain of actual coverage minus the total maintenance cost is maximal.



79	18	3	6	8	6	2	9	17	24
99	20	4	8	12	15	8	16	19	17
46	60	5	9	62	30	38	21	15	19
53	31	21	13	100	90	17	16	20	28
70	83	98	70	68	67	30	29	23	28
22	44	88	77	1	83	17	56	51	58
78	35	52	38	36	42	66	50	53	11
77	56	75	52	47	37	20	38	35	23
67	67	100	67	91	58	32	21	23	27
46	71	81	54	74	87	67	49	34	25

White numbers: expected gains; yellow squares: placed antennae

We now show that we can pre-compute a 3d array with the net gain for each possible antenna placement and type, making it much easier to express the objective function and much faster to solve problem instances.

Model without Pre-Computation

```
...
set of int: Zones = 1..z; % the region has z by z zones
array[Zones,Zones] of int: ExpGain =
   [|79,18,3,6,8,6,2,9,17,24|...|]; % expected gain per zone
set of int: Types = 1..t; % type i covers i by i zones
array[Types] of int: Cost; % maintenance costs
set of int: Antennae = 1..antennae;
% Variables (for lower-left coordinates) and constraints:
array[Antennae] of var Zones: X; % X[a] = x-coordinate of a
array[Antennae] of var Zones: Y; % Y[a] = y-coordinate of a
array[Antennae] of var Types: Type; % Type[a] = type of a
constraint diffn(X,Y,Type,Type); % no coverage overlaps
constraint forall(a in Antennae)
   (X[a]+Type[a] <= z+1  /\  Y[a]+Type[a] <= z+1);
% Objective:
array[Antennae] of var int: Gain; % Gain[a] = gain of a
constraint forall(a in Antennae)(Gain[a]=sum(x,y in Zones)
   (ExpGain[x,y] * bool2int(X[a] <= x  /\  x < X[a]+Type[a]
   /\  Y[a] <= y  /\  y < Y[a]+Type[a])));
var int: cost=sum(a in Antennae)(Cost[Type[a]]);% total cost
solve maximize sum(Gain) - cost;
```

Model with Pre-Computation

```
12 ... % lines 1 to 12 of the previous model
13 % Pre-computation:
14 array[Zones,Zones,Types] of int: NetGain =
     array3d(Zones,Zones,Types, [sum(w,h in 0..Type[t]-1
     where x+w <= z /\ y+h <= z)(ExpGain[x+w,y+h]) - Cost[t]
     | x,y in Zones, t in Types]);
15 % Objective:
16 solve maximize sum(a in Antennae)
    (NetGain[X[a],Y[a],Type[a]]);
```

This model yields better inference and faster solving.

Solving to optimality with Gecode (CP) for `z=10`:

pre-computation	`antennae`	`types`	# nodes	seconds
without	1	4	927	0.007
with	1	4	65	0.001
without	2	4	11,445,833	106.936
with	2	4	361	0.005
without	3	4	timeout	timeout
with	3	4	961	0.015
with	5	4	188,844	2.642

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 warehouse.

Determine which warehouses to open, and which of them should supply the various shops, so that:

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

WLP: Sample Instance Data

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

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

$$\texttt{maintCost} = 30$$

$\texttt{Capacity} =$

Berlin	London	Ankara	Paris	Rome
1	4	2	1	3

$\texttt{SupplyCost} =$

	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

WLP Model 1: Decision Variables

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

`Supplier` =

Shop_1	Shop_2	$\cdots$	Shop_{10}
∈ `Whs`	∈ `Whs`	$\cdots$	∈ `Whs`

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

Redundant decision variables:

`Open` =

Berlin	London	Ankara	Paris	Rome
∈ `0..1`	∈ `0..1`	∈ `0..1`	∈ `0..1`	∈ `0..1`

`Open[w]=1` if and only if (iff) warehouse `w` is opened.

☞ Our chosen array names always reflect 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 index `Supplier[s]` is a decision variable.

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

WLP Model 1: Channelling Constraint

One-way channelling constraint from the `Supplier` variables to the redundant `Open` variables:

```
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]` is a decision variable.

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

WLP Model 1: Channelling Constraint

Alternative: Two-way channelling constraint between the `Supplier` variables and the redundant `Open` variables:

```
forall(w in Whs)
 (Open[w] = bool2int(exists(s in Shops)(Supplier[s]=w)))
```

A warehouse is opened iff there exists a shop 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.

Also consider making `Open` an array of Boolean variables and using `bool2int` for its summation in the objective.

WLP Model 1: Capacity Constraint

Capacity constraint (2):

```
global_cardinality_low_up_closed
(Supplier, Whs, [0 | i in Whs], Capacity)
```

Each warehouse is a supplier of a number of shops at most equal to its capacity.

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 Whs)(bool2int(
   exists(s in Shops)(Supplier[s]=w)))
```

We can also use the `nvalue` constrained function:

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

This model 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 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` iff 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 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 Whs)
     (Supply[s,w] * SupplyCost[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 parameter `SupplyCost[s,w]` is weighted by the decision variable `Supply[s,w]`.

WLP Model 3: Constraints

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

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

Each shop is supplied by exactly one warehouse.

WLP Model 3: Constraints (end)

Capacity constraint (2), in isolation:

```
forall(w in Whs)
  (sum(Supply[..,w]) <= Capacity[w])
```

Two-way channelling constraint, in isolation:

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

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

```
forall(w in Whs)
  (forall(s in Shops)(Supply[s,w]<=Open[w]))
```

Capacity (2) & one-way channelling constraints combined:

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

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

Outline

The Sport Scheduling Problem (SSP)

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

- $|\texttt{Teams}| = \texttt{n}$
- $|\texttt{Weeks}| = \texttt{n-1}$
- $|\texttt{Periods}| = \texttt{n div 2}$

subject to the following constraints:

1. Each 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 `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

UPPSALA UNIVERSITET

The Sport Scheduling Problem (SSP)

Find schedule in `Periods` × `Weeks` → `Teams` × `Teams` for

- $|$`Teams`$|$ = `n`
- $|$`Weeks`$|$ = `n-1`
- $|$`Periods`$|$ = `n div 2`

subject to the following constraints:

1. Each 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 `n=8`,
with a dummy week `n` of duplicate games:

	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;assert(n>1 /\ n mod 2 =0,"Odd n")`

Useful Ranges and Sets:

- `Teams = 1..n`
- `Weeks = 1..(n-1)`
- `ExtendedWeeks = 1..n`
- `Periods = 1..(n div 2)`
- `Slots = 1..2`
- `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

A 3d matrix `Team[Periods,ExtendedWeeks,Slots]` of 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 transforming constraint (3) into:

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

Predicate `global_cardinality_low_up_closed` need not be used and can be replaced by a stronger predicate.

`Team` =

$$\begin{array}{c|c|c|c|c|c|c|c|c|}
 & \multicolumn{2}{c}{\text{Wk } 1} & \multicolumn{2}{c}{\cdots} & \multicolumn{2}{c}{\text{Wk } n-1} & \multicolumn{2}{c}{\text{Wk } n} \\
 & 1 & 2 & \cdots & \cdots & 1 & 2 & 1 & 2 \\
\hline
\text{P } 1 & \in \mathtt{T} & \in \mathtt{T} & \cdots & \cdots & \in \mathtt{T} & \in \mathtt{T} & \in \mathtt{T} & \in \mathtt{T} \\
\vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\
\text{P } n/2 & \in \mathtt{T} & \in \mathtt{T} & \cdots & \cdots & \in \mathtt{T} & \in \mathtt{T} & \in \mathtt{T} & \in \mathtt{T} \\
\hline
\end{array}$$

`Team[p,w,s]` is the number of the team that plays in period `p` of week `w` in game slot `s`.

SSP Model 1: Constraints

Twice-per-period constraint (3’):

```
forall(p in Periods)
 (global_cardinality_closed
   (array1d(Team[p,..,..]), Teams, [2 | i in 1..n]))
```

In each period, each team number occurs exactly twice within the slots of the weeks in `Team`.

Once-per-week constraint (2):

```
forall(w in ExtendedWeeks)
 (alldifferent(Team[..,w,..]))  % works on 2d slices
```

In each week, incl. the dummy week, there are no duplicate team numbers 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`!

Declare a 2d matrix `Game[Periods,Weeks]` of redundant decision variables in `Games` over the non-extended weeks:

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

`Game[p,w]` is the game played in period `p` of week `w`.

SSP Model 1: Constraints (end)

Each-game-once constraint (1):

```
alldifferent(Game)
```

There are no duplicate game numbers in `Game`.

Channelling constraint (alternatively use `table`:
☞ see Topic 8: Inference & Search in CP & LCG):

```
forall(p in Periods, w in Weeks)
  (Team[p,w,1]*n+Team[p,w,2] = Game[p,w])
```

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

Constraints (2) and (3') are hard to formulate using `Game`.

SSP Model 2: Smaller Domains for `Game`

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

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

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

and

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

The constraints (1) and (2) are now automatically enforced: need to determine the period of each game, not its week!

Interested in More Details?

For more details on WLP & 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.