

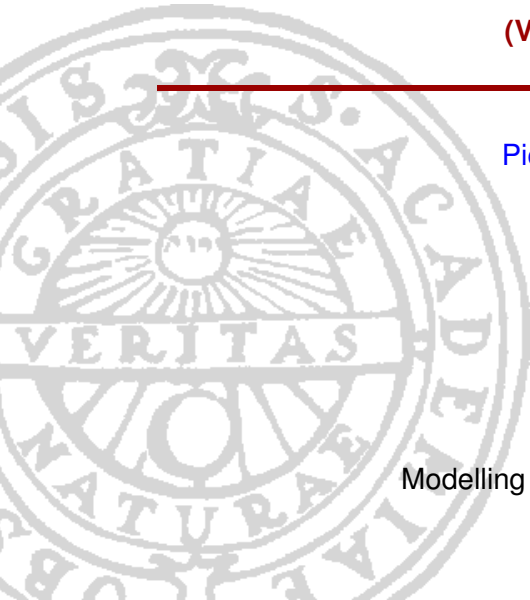
Topic 6: Case Studies

(Version of 26th October 2017)

Pierre Flener and Gustav Björdal

ASTRA Research Group
on Combinatorial Optimisation
Uppsala University
Sweden

Course 1DL448:
Modelling for Combinatorial Optimisation





Outline

Black-Hole
Patience

Warehouse
Location

Sport
Scheduling

1. Black-Hole Patience

2. Warehouse Location

3. Sport Scheduling



Outline

Black-Hole
Patience

Warehouse
Location

Sport
Scheduling

1. Black-Hole Patience

2. Warehouse Location

3. Sport Scheduling



Example (Black-Hole Patience)



Move all the cards into the black hole. A fan top card can be moved if it is a rank apart from the black-hole top card, independently of suit (\spadesuit , \clubsuit , \diamondsuit , \heartsuit); aces (A,1) and kings (K,13) are a rank apart.

Card encoding: \spadesuit : 1..13, \clubsuit : 14..26, \diamondsuit : 27..39, \heartsuit : 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` constraints 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|]);
```



Example (Black-Hole Patience: model)



Move all the cards into the black hole. A fan top card can be moved if it is a rank apart from the black-hole top card, independently of suit (\spadesuit , \clubsuit , \diamondsuit , \heartsuit); aces (A,1) and kings (K,13) are a rank apart.

Let $\text{Card}[p]$ denote the card at position p in the black hole:

```
Card[1]=1/\forall (p in 1..51) (rankApart(Card[p],Card[p+1]))
```

Using the Card vars, how to model the other constraint?

Let $\text{Pos}[c]$ denote the position of card c in the black hole:

```
Pos[1]=1/\forall ("card c1 on top of c2 in a fan") (Pos[c1]<Pos[c2])
```

Using the Pos variables, how to model the other constraint?



Example (Black-Hole Patience: channelling)

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

Observe that: $\forall c, p \in 1..52 : \text{Card}[p]=c \Leftrightarrow \text{Pos}[c]=p$.

Or, equivalently: $\forall c \in 1..52 : \text{Card}[\text{Pos}[c]]=c$.

Seen as functions, `Card` and `Pos` are each other's inverse!

This is the semantics of `inverse` (`Card`, `Pos`).

This semantics implies the `alldifferent` (`Card`) and `alldifferent` (`Pos`) constraints that are still missing, so we do not need to add them to the model.



Outline

Black-Hole
Patience

Warehouse
Location

Sport
Scheduling

1. Black-Hole Patience

2. Warehouse Location

3. Sport Scheduling



Example (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

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

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

$$\text{maintCost} = 30$$

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
⋮	⋮	⋮	⋮	⋮	⋮
Shop ₁₀	47	65	55	71	95

There is an optimal solution where Shop₄ is supplied by Rome, not by Berlin.



WLP Model 1: Decision Variables

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

$$\text{Supplier} = \begin{array}{cccc} \text{Shop}_1 & \text{Shop}_2 & \dots & \text{Shop}_{10} \\ \hline \in \text{Whs} & \in \text{Whs} & \dots & \in \text{Whs} \end{array}$$

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

Redundant decision variables:

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

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

☞ Our chosen array names always reflect **functions**.



WLP Model 1: Objective

```
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.

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, but is ten times faster for the given instance data, using CP or LCG solvers.

**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):

		Berlin	London	Ankara	Paris	Rome
Supply =	Shop ₁	∈ 0..1	∈ 0..1	∈ 0..1	∈ 0..1	∈ 0..1
	⋮	⋮	⋮	⋮	⋮	⋮
	Shop ₁₀	∈ 0..1	∈ 0..1	∈ 0..1	∈ 0..1	∈ 0..1

Supply[s, w] = 1 iff shop s is supplied by warehouse w .

Redundant decision variables (as in Model 1):

	Berlin	London	Ankara	Paris	Rome
Open =	∈ 0..1	∈ 0..1	∈ 0..1	∈ 0..1	∈ 0..1

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



WLP Model 3: Objective

The objective function is linear:

```
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 $\text{SupplyCost}[s,w]$ is weighted by the decision variable $\text{Supply}[s,w]$.



WLP Model 3: Constraints

Total-function constraint (1):

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

Each shop is supplied by exactly one warehouse.



WLP Model 3: Constraints (end)

Capacity constraint (2), in isolation:

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

Two-way channelling constraint, in isolation:

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

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



Outline

Black-Hole
Patience

Warehouse
Location

Sport
Scheduling

1. Black-Hole Patience

2. Warehouse Location

3. Sport Scheduling



Example (The Sport Scheduling Problem, SSP)

Find schedule in $\text{Periods} \times \text{Weeks} \rightarrow \text{Teams} \times \text{Teams}$ for

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

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



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 $\text{Team}[\text{Periods}, \text{ExtendedWeeks}, \text{Slots}]$ 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.

$\text{Team} =$

	Wk 1		...		Wk $n-1$		Wk n	
	1	2	1	2	1	2
P 1	$\in \text{T}$	$\in \text{T}$	$\in \text{T}$	$\in \text{T}$	$\in \text{T}$	$\in \text{T}$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
P $n/2$	$\in \text{T}$	$\in \text{T}$	$\in \text{T}$	$\in \text{T}$	$\in \text{T}$	$\in \text{T}$

$\text{Team}[p, w, s]$ 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'), in future MiniZinc syntax:

```
forall(p in Periods)
  (global_cardinality_closed
   (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), in future MiniZinc syntax:

```
forall(w in ExtendedWeeks)
  (alldifferent (Team[., w, ..]))
```

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:

		Week 1	...	Week $n - 1$
<code>Game =</code>	Period 1	<code>∈ Games</code>	<code>...</code>	<code>∈ Games</code>
	<code>⋮</code>	<code>⋮</code>	<code>⋮</code>	<code>⋮</code>
	Period $n/2$	<code>∈ Games</code>	<code>...</code>	<code>∈ Games</code>

`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`):

```
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 Régim, Jean-Charles.

[Constraint programming in OPL.](#)

PPDP 1999, pages 98–116. *Lecture Notes in Computer Science 1702*. Springer-Verlag, 1999.