Topic 6: Case Studies  
(Version of 5th August 2022)  

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

1. Black-Hole Patience

2. Cost-Aware Scheduling

3. Warehouse Location

4. Sport Scheduling
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.


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

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

Defining a help predicate and avoiding `mod` on variables:

```plaintext
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:

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

Let us model “adjacent black-hole cards are a rank apart”.

- 4 -
Model: Decision Variables and Constraints

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

3. constraint $\text{Card}[1] = 1$; % the card at position 1 is A♠
4. constraint $\forall (p \in 1..51) (\text{rankApart}(\text{Card}[p], \text{Card}[p+1]))$;

Let us model “black-hole cards respect the order in fans”:

5. constraint $\forall (\text{“fan with card” c1 “on top of” c2 “on top of” c3})$
6. (let { var 2..52: p1; var 2..52: p2; var 2..52: p3 } in $\text{Card}[p1]=c1/\text{Card}[p2]=c2/\text{Card}[p3]=c3/\text{p1}<p2/\text{p2}<p3$);
7. % constraint all_different(\text{Card}); % implied by correct data!

or, equivalently, without implicit element constraints:

6. (\text{value_precede_chain}([c1,c2,c3],\text{Card}));

Let us now formulate that second constraint even better.
Model: Redundant Variables & Channelling

Let $\text{Pos}[c]$ denote the position of card $c$ in the black hole. The black-hole cards respect the order in the given fans:

5. $\text{constraint} \ \text{Pos}[1] = 1; \ % \ the \ position \ of \ card \ A♠ \ is \ 1$
6. $\text{constraint} \ \forall \text{c1, c2, c3} \ (\text{fan \ with \ card} \ c1 \ \text{on \ top \ of} \ c2 \ \text{on \ top \ of} \ c3)$
7. $(\text{Pos}[c1] < \text{Pos}[c2] \land \text{Pos}[c2] < \text{Pos}[c3]);$
8. $\% \ \text{constraint} \ \text{all_different(Pos);} \ % \ implied \ by \ correct \ data!$

How to model “adjacent black-hole cards are a rank apart” with the $\text{Pos}[c]$ variables?! Let us use the latter together with the $\text{Card}[p]$ variables, and channel between them.

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

8. $\text{constraint} \ \text{inverse(Card,Pos);}$
   $\% \ implies \ \text{all_different(Card)} \lor \text{all_different(Pos)}$

The model with mutually redundant variables and the 2-way channelling constraint is much faster (at least on a CP or LCG solver) than the models with only the $\text{Card}$ variables.
Outline

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

Consider the core of CSPlib problem 059. Given are:

- Machines, each with capacitated reusable resources.
- Jobs, each with a duration, earliest start and latest end times, a consumption of energy (which is an overall consumable resource, not a machine-specific reusable one), and requirements for the reusable resources.
- A time horizon, each step with 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

1. `enum Resources; % say: {cpu,ram,io};`
2. `int: nMachines; set of int: Machines = 1..nMachines;`
3. `array[Machines,Resources] of int: Capacity;`
4. `int: nTimeSteps; % say: 288, for every 5 minutes over 24h`
5. `set of int: Times = 0..nTimeSteps; % time points`
6. `set of int: Steps = 1..nTimeSteps; % step s is from s-1 to s`
7. `array[Steps] of float: EnergyCost; % EnergyCost[s] €/kW in s`
8. `int: nJobs; set of int: Jobs = 1..nJobs;`
9. `array[Jobs] of Steps: Duration; % j lasts Duration[j] steps`
10. `array[Jobs] of Times: EarliestS; % j starts >= EarliestS[j]`
11. `array[Jobs] of Times: LatestEnd; % j ends <= LatestEnd[j]`
13. `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]
```

So a job of 6 time steps and 1151 kWh starting at time 118 costs 
\[1151 \cdot (0.04732 \cdot 2 + 0.08093 \cdot 4)\] = 481€, and 
\[1151 \cdot (0.08093 \cdot 4 + 0.08619 \cdot 2)\] = 571€ at time 122.
array[Jobs] of var Times: Start; % j starts at time Start[j]
array[Jobs] of var Machines: Machine; % 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 has energy cost Cost[j]
... % see the next slide!
solve minimize sum(Cost) + ...;
Defining the variables $\text{Cost}[j]$ without precomputation:

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

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

Defining the variables $\text{Cost}[j]$ with precomputation:

```prolog
% 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 compiles for Gecode in a blink into only 343 KB of code with 100 constraints and 100 variables, and a feasible solution is found six times faster.
Outline

1. Black-Hole Patience

2. Cost-Aware Scheduling

3. Warehouse Location

4. Sport Scheduling
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 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 at most equal to its capacity.
3. The sum of the actually incurred maintenance costs and supply costs is minimal.
### WLP: Sample Instance Data

Shops = \{Shop_1, Shop_2, \ldots, Shop_{10}\}


\text{maintCost} = 30

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Berlin</th>
<th>London</th>
<th>Ankara</th>
<th>Paris</th>
<th>Rome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shop _1</th>
<th>Berlin</th>
<th>London</th>
<th>Ankara</th>
<th>Paris</th>
<th>Rome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>24</td>
<td>11</td>
<td>25</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shop _2</th>
<th>Berlin</th>
<th>London</th>
<th>Ankara</th>
<th>Paris</th>
<th>Rome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>28</td>
<td>27</td>
<td>82</td>
<td>83</td>
<td>74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shop _3</th>
<th>Berlin</th>
<th>London</th>
<th>Ankara</th>
<th>Paris</th>
<th>Rome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>74</td>
<td>97</td>
<td>71</td>
<td>96</td>
<td>70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shop _4</th>
<th>Berlin</th>
<th>London</th>
<th>Ankara</th>
<th>Paris</th>
<th>Rome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>55</td>
<td>73</td>
<td>69</td>
<td>61</td>
</tr>
</tbody>
</table>

| \ldots   | \ldots | \ldots | \ldots | \ldots | \ldots |

<table>
<thead>
<tr>
<th>Shop _10</th>
<th>Berlin</th>
<th>London</th>
<th>Ankara</th>
<th>Paris</th>
<th>Rome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>47</td>
<td>65</td>
<td>55</td>
<td>71</td>
<td>95</td>
</tr>
</tbody>
</table>
WLP Model 1: Decision Variables

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

\[
\text{Supplier} = \begin{bmatrix}
\text{Shop}_1 & \text{Shop}_2 & \cdots & \text{Shop}_{10} \\
\in \text{Whs} & \in \text{Whs} & \cdots & \in \text{Whs}
\end{bmatrix}
\]

\text{Supplier}[s] \text{ denotes the supplier warehouse for shop } s.

Variables redundant with \text{Supplier}, but not mutually:

\[
\text{Open} = \begin{bmatrix}
\text{Berlin} & \text{London} & \text{Ankara} & \text{Paris} & \text{Rome} \\
\in 0..1 & \in 0..1 & \in 0..1 & \in 0..1 & \in 0..1
\end{bmatrix}
\]

\text{Open}[w] = 1 \text{ if and only if (iff) warehouse } w \text{ is opened.}

☞ Our chosen array names always reflect total functions.
WLP Model 1: Objective

\[
\text{solve minimize } \text{maintCost} \times \sum (\text{Open}) + \sum (s \in \text{Shops})(\text{SupplyCost}[s, \text{Supplier}[s]])
\]

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 \( \text{MaintCost}[w] \), then the first term becomes

\[
\sum (w \in \text{Whs})(\text{MaintCost}[w] \times \text{Open}[w]).
\]
**WLP Model 1: Channelling Constraint**

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

\[ \forall s \in \text{Shops} \left( \text{Open}[\text{Supplier}[s]] = 1 \right) \]

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 its redundant Open variables:

\[
\text{forall}(w \text{ in Whs}) \\
(\text{Open}[w] = (\exists s \text{ in Shops})(\text{Supplier}[s]=w))
\]

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.

Nothing changes if Open is an array of Boolean variables.
WLP Model 1: Capacity Constraint

Capacity constraint (2):

\[
\text{global_cardinality_low_up_closed} \\
(\text{Supplier, Whs, [0 | w in Whs]}, \text{Capacity})
\]

Each actually opened warehouse is a supplier of a number of shops at most equal to its 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:

\[
\text{maintCost} \times \\
\sum_{w \in \text{Whs}} (\exists s \in \text{Shops})(\text{Supplier}[s]=w)
\]

We can alternatively use the nvalue constrained function:

\[
\text{maintCost} \times \\
n\text{value}(\text{Supplier})
\]

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 total-function constraint (1):

\[
\text{Supply} = \begin{array}{cccccc}
\text{Shop}_1 & \text{Berlin} & \text{London} & \text{Ankara} & \text{Paris} & \text{Rome} \\
\vdots & \in 0..1 & \in 0..1 & \in 0..1 & \in 0..1 & \in 0..1 \\
\text{Shop}_{10} & \vdots & \vdots & \vdots & \vdots & \vdots \\
\in 0..1 & \in 0..1 & \in 0..1 & \in 0..1 & \in 0..1 \\
\end{array}
\]

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

Redundant decision variables (as in Model 1):

\[
\text{Open} = \begin{array}{cccccc}
\text{Berlin} & \text{London} & \text{Ankara} & \text{Paris} & \text{Rome} \\
\in 0..1 & \in 0..1 & \in 0..1 & \in 0..1 & \in 0..1 \\
\end{array}
\]

Open \([w] = 1\) if and only if warehouse \(w\) is opened.
WLP Model 3: Objective

The objective can now be expressed in linear fashion:

\[
\text{solve minimize} \quad \text{maintCost} \times \text{sum}(\text{Open}) \\
+ \quad \text{sum}(s \text{ in Shops, } w \text{ in Whs}) \\
\quad (\text{Supply}[s,w] \times \text{SupplyCost}[s,w])
\]

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 \(\text{Supply}[s,w]\).
The total-function constraint (1) now needs to be modelled, and can be expressed in linear fashion without $\text{count}$:

$$\forall (s \text{ in } \text{Shops}) \left( \sum (\text{Supply}[s,..]) = 1 \right)$$

Each shop is supplied by exactly one actually opened warehouse.
WLP Model 3: Constraints (end)

Capacity constraint (2), in isolation:

\[ \text{forall} \( w \in \text{Whs} \) \]
\[ (\text{sum}(\text{Supply}[\ldots, w]) \leq \text{Capacity}[w]) \]

Two-way channelling constraint, in isolation:

\[ \text{forall} \( w \in \text{Whs} \) \]
\[ (\text{sum}(\text{Supply}[\ldots, w]) > 0 \leftrightarrow \text{Open}[w] = 1) \]

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

\[ \text{forall} \( w \in \text{Whs} \) \]
\[ (\text{forall} \( s \in \text{Shops} \) (\text{Supply}[s, w] \leq \text{Open}[w])) \]

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

\[ \text{forall} \( w \in \text{Whs} \) \]
\[ (\text{sum}(\text{Supply}[\ldots, w]) \leq \text{Capacity}[w] \times \text{Open}[w]) \]

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
The Sport Scheduling Problem (SSP)

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

- $|\text{Teams}| = n$ and $n$ is even
- $|\text{Weeks}| = n-1$
- $|\text{Periods}| = n/2$ 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 $n=8$

<table>
<thead>
<tr>
<th></th>
<th>Wk 1</th>
<th>Wk 2</th>
<th>Wk 3</th>
<th>Wk 4</th>
<th>Wk 5</th>
<th>Wk 6</th>
<th>Wk 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>P 1</td>
<td>1 vs 2</td>
<td>1 vs 3</td>
<td>2 vs 6</td>
<td>3 vs 5</td>
<td>4 vs 7</td>
<td>4 vs 8</td>
<td>5 vs 8</td>
</tr>
<tr>
<td>P 2</td>
<td>3 vs 4</td>
<td>2 vs 8</td>
<td>1 vs 7</td>
<td>6 vs 7</td>
<td>6 vs 8</td>
<td>2 vs 5</td>
<td>1 vs 4</td>
</tr>
<tr>
<td>P 3</td>
<td>5 vs 6</td>
<td>4 vs 6</td>
<td>3 vs 8</td>
<td>1 vs 8</td>
<td>1 vs 5</td>
<td>3 vs 7</td>
<td>2 vs 7</td>
</tr>
<tr>
<td>P 4</td>
<td>7 vs 8</td>
<td>5 vs 7</td>
<td>4 vs 5</td>
<td>2 vs 4</td>
<td>2 vs 3</td>
<td>1 vs 6</td>
<td>3 vs 6</td>
</tr>
</tbody>
</table>
The Sport Scheduling Problem (SSP)

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

- $|\text{Teams}| = n$ and $n$ is even
- $|\text{Weeks}| = n-1$
- $|\text{Periods}| = n/2$ 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 $n=8$, with a dummy week $n$ of duplicate games:

<table>
<thead>
<tr>
<th>Wk 1</th>
<th>Wk 2</th>
<th>Wk 3</th>
<th>Wk 4</th>
<th>Wk 5</th>
<th>Wk 6</th>
<th>Wk 7</th>
<th>Wk 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>P 1</td>
<td>1 vs 2</td>
<td>1 vs 3</td>
<td>2 vs 6</td>
<td>3 vs 5</td>
<td>4 vs 7</td>
<td>4 vs 8</td>
<td>5 vs 8</td>
</tr>
<tr>
<td>P 2</td>
<td>3 vs 4</td>
<td>2 vs 8</td>
<td>1 vs 7</td>
<td>6 vs 7</td>
<td>6 vs 8</td>
<td>2 vs 5</td>
<td>1 vs 4</td>
</tr>
<tr>
<td>P 3</td>
<td>5 vs 6</td>
<td>4 vs 6</td>
<td>3 vs 8</td>
<td>1 vs 8</td>
<td>1 vs 5</td>
<td>3 vs 7</td>
<td>2 vs 7</td>
</tr>
<tr>
<td>P 4</td>
<td>7 vs 8</td>
<td>5 vs 7</td>
<td>4 vs 5</td>
<td>2 vs 4</td>
<td>2 vs 3</td>
<td>1 vs 6</td>
<td>3 vs 6</td>
</tr>
</tbody>
</table>
SSP Model 1: Data

Parameter:

- int: n; assert(n>1 \&\& 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

A 3d matrix $\text{Team}[\text{Periods, ExtendedWeeks, Slots}]$ of variables in $\text{Teams}$, 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 $\text{global_cardinality_low_up_closed}$ need not be used and can be replaced by a stronger predicate.

Team =

<table>
<thead>
<tr>
<th></th>
<th>Wk 1</th>
<th></th>
<th>Wk 2</th>
<th></th>
<th>Wk $n-1$</th>
<th></th>
<th>Wk $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_{1}$</td>
<td>$\in T$</td>
<td>$\in T$</td>
<td>$\in T$</td>
<td>$\in T$</td>
<td>$\in T$</td>
<td>$\in T$</td>
<td>$\in T$</td>
</tr>
<tr>
<td>$P_{n/2}$</td>
<td>$\in T$</td>
<td>$\in T$</td>
<td>$\in T$</td>
<td>$\in T$</td>
<td>$\in T$</td>
<td>$\in T$</td>
<td>$\in T$</td>
</tr>
</tbody>
</table>

$\text{Team}[p, w, s]$ is the numeric name of the team that plays in period $p$ of week $w$ in game slot $s$. 
SSP Model 1: Constraints

Twice-per-period constraint (3’):

\[
\text{forall}(p \in \text{Periods}) \\
\quad (\text{global_cardinality_closed} \\
\quad \quad (\text{Team}[p,\ldots,\ldots], \text{Teams}, \{2 \mid i \in 1..n\}))
\]

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

Once-per-week constraint (2):

\[
\text{forall}(w \in \text{ExtendedWeeks}) \\
\quad (\text{all_different}(\text{Team}[\ldots,w,\ldots]))
\]

In each week, incl. the dummy week, there are no duplicate team names 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 $\text{Game}[\text{Periods}, \text{Weeks}]$ of decision variables in Games over the non-extended weeks:

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

$\text{Game}[p, w]$ is the game played in period $p$ of week $w$.

The 2d $\text{Game}$ is mutually redundant with the first $n - 1$ 2d columns of the 3d $\text{Team}$, which is over the extended weeks.
SSP Model 1: Constraints (end)

Each-game-once constraint (1):

\[\text{all\_different}(\text{Game})\]

There are no duplicate game numbers in \text{Game}.

Two-way channelling constraint (but rather use \text{table}:

\[\text{forall}(p \text{ in } \text{Periods}, \ w \text{ in } \text{Weeks})\]

\[\text{Team}[p, w, \text{one}] \times n + \text{Team}[p, w, \text{two}] = \text{Game}[p, w]\]

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

Constraints (2) and (3’) are hard to formulate using \text{Game}.

Add the symmetry-breaking constraints of slide 29 of Topic 5: Symmetry.
SSP Model 2: Smaller Domains for Game

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 \( f \text{ vs } s \) of the previous week into a game \( f' \text{ vs } s' \), where

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

The constraints (1) and (2) are now automatically enforced: we must 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. 

Van Hentenryck, Pascal. 
Constraint and integer programming in OPL. 

Van Hentenryck, Pascal; Michel, Laurent; Perron, Laurent; and Régis, Jean-Charles. 
Constraint programming in OPL. 