Topic 8: Inference & Search in CP & LCG

(Version of 14th November 2018)

Pierre Flener and Jean-Noël Monette

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

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Outline

1. Annotations

2. Inference Annotations for CP & LCG

3. Search Annotations for CP & LCG

4. Case Studies

- Balanced Incomplete Block Design
- Warehouse Location
- Sport Scheduling

UPPSALA UNIVERSITET

Annotations:

- Annotations provide information to the backend or to the MiniZinc-to-FlatZinc compiler.
- Annotations are optional.
- A backend may ignore any of the annotations.
- The compiler may introduce further annotations.
- Annotations are attached with `::` to model items.
  Example: `var int: x :: is_defined_var;`
- Annotations do not affect the model semantics.

Annotations to a variable declaration:

- `is_defined_var`
  The variable is functionally defined by some constraint.
- `var_is_introduced`
  The variable has been introduced by the compiler.

Annotation to a function definition:

- `promise_total`
  The function is total.

Annotations to a constraint:

- `defines_var(x)`
  Variable `x` is functionally defined by this constraint.
  Example: `constraint x = A[z] :: defines_var(x);`
  This can be exploited, say by a CBLS backend.
- Other annotations suggest a propagator to use for the constraint by a CP or LCG backend: see slide 9.

Annotations to the objective:

- Annotations suggest a search strategy to use by a CP or LCG backend: see slide 15.

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Domains (reminder)

Definition

The domain of a variable v, denoted by $\mathrm{dom}(v)$, is the set of values that v can still take during search:

- The domains of the variables are reduced by search and by inference (see the next two slides).
- A variable is said to be fixed if its domain is a singleton.
- A model is unsatisfiable if a variable domain is empty.

Note the difference between:

- a domain as a technology-independent declarative entity at the modelling level; and
- a domain as a procedural data structure for CP solving.

UPPSALA UNIVERSITET

CP Solving (reminder)

Tree Search:

Satisfaction problem:

1. At the root, set each variable domain as in the model.
2. Perform inference (see the next slide).
3. If some variable domain is empty, then backtrack.
4. If all variables are fixed, then we have a solution.
5. Pick an unfixed variable v, partition its domain into two parts π_1 and π_2, and make two branches: one with $v \in \pi_1$, and the other one with $v \in \pi_2$.
6. Recursively explore each of the two branches.

Optimisation problem: when a solution is found, add the constraint that the next solution must have a better objective value (see step 3 of branch-and-bound for IP).

CP Inference

Definition

A propagator for a predicate γ removes from the current domains of the variables of a γ-constraint ~~the~~ values that cannot be part of a solution to that constraint.

Not all impossible values need to be removed:

- A domain-consistency propagator removes all impossible values from the domains.
- A bounds-consistency propagator only removes all impossible min and max values from the domains.

There exist other, unnamed consistencies for propagators.

There is a trade-off between the time & space complexity of a propagator and its achieved removal of domain values.

Example (Linear equality constraints)

Consider the linear constraint `3 * x + 4 * y = z` with dom(`x`) = `0..1` = dom(`y`) and dom(`z`) = `0..10`:

- A bounds-consistency propagator reduces dom(`z`) to `0..7`.
- A domain-consistency propagator reduces dom(`z`) to `{0,3,4,7}`.

Time complexity:

- A bounds-consistency propagator for a linear equality constraint can be implemented to run in $\mathcal{O}(n)$ time, where n is the number of variables in the constraint.
- A domain-consistency propagator for a linear equality constraint can be implemented to run in $\mathcal{O}(n \cdot d^2)$ time, where n is the number of variables in the constraint and d is the sum of their domain sizes, hence in time pseudo-polynomial = exponential in input magnitude.

UPPSALA UNIVERSITET

Controlling the CP Inference

The choice of the right propagator for each constraint may be critical for performance.

Each CP solver and LCG solver has a default propagator for each available constraint predicate.

It is possible to override the defaults with annotations:

- `::` `domain` asks for a domain-consistency propagator.
- `::` `bounds` asks for a bounds-consistency propagator.

Annotations may be ignored, only partially followed, or just approximated: annotations are just suggestions.

Example (n-Queens)

```
1 array[1..n] of var 1..n: Row;
2 constraint alldifferent(Row)                :: domain;
3 constraint alldifferent
4     ([ Row[q]+q           | q in 1..n]) :: domain;
5 constraint alldifferent
6     ([ Row[q]-q           | q in 1..n]) :: domain;
```

Test results with Gecode (CP) to first solution for n=101:

inference	# nodes	seconds
default (no annotation)	348,193	5.5
bounds on alldifferent	348,193	5.5
domain on alldifferent	209,320	3.2

Example (n-Queens)

```
1 array[1..n] of var 1..n: Row;
2 constraint alldifferent(Row)                :: domain;
3 constraint alldifferent
4     ([(Row[q]+q)::bounds | q in 1..n]) :: domain;
5 constraint alldifferent
6     ([(Row[q]-q)::bounds | q in 1..n]) :: domain;
```

Test results with Gecode (CP) to first solution for `n=101`:

inference	# nodes	seconds
default (no annotation)	348,193	5.5
`bounds` on `alldifferent`	348,193	5.5
`domain` on `alldifferent`	209,320	3.2
+ `bounds` on the linear constraints	$>$ 20M	$>$ 600.0
`bounds` on all the constraints	$>$ 20M	$>$ 600.0

Asking for bounds consistency on the implicit linear equality constraints backfires here, as each is on only 2 variables, but it may pay off upon more variables (and be default then).

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Search Strategies

Search Strategies:

- On which variable to branch next?
- How to split the current domain of the chosen variable?
- Which search (depth-first, breadth-first, . . .) to use?

The strategy is usually depth-first left-to-right search.

One can suggest to a CP or LCG backend on which variable to branch and how, by making an annotation with:

- a variable selection strategy, and
- a value selection strategy.

Variable Selection Strategy

The variable selection strategy has an impact on the size of the search tree, especially if the constraints are processed with propagation at every node of the search tree or if the whole search tree is explored: for example, when it is an optimisation problem or when there are no solutions.

Example (Impact of the variable selection strategy)

Consider `var 1..2: x`, `var 1..4: y`, `var 1..6: z`, branching on all domain values, but no constraints:

- If selecting the variables in the order `x`, `y`, `z`, then the CP search tree has $1 + 2 + 2 \cdot 4 + 2 \cdot 4 \cdot 6 = 59$ nodes and $2 \cdot 4 \cdot 6 = 48$ leaves.
- If selecting the variables in the order `z`, `y`, `x`, then the CP search tree has $1 + 6 + 6 \cdot 4 + 6 \cdot 4 \cdot 2 = 79$ nodes and also $6 \cdot 4 \cdot 2 = 48$ leaves.

Definition (First-Fail Principle)

To succeed, first try where you are most likely to fail.
In practice:

- Pick a variable with the smallest current domain.
- Pick a variable involved in the largest #constraints.
- Pick a variable causing the largest #recent backtracks.

Example (Impact of the variable selection strategy)

Finding the first solution to `101`-queens with Gecode (CP):

search	# nodes	seconds
default (no annotation)	348,193	5.5
`first_fail`	323,275	5.3
`anti_first_fail`	$>$ 20M	$>$ 600.0
`input_order`	$>$ 13M	$>$ 600.0

(Continued on slide 19)

Value Selection Strategy

The value selection strategy has an impact on the size of the search tree when optimising, when only searching for the first solution, or when performing incomplete search (say when using a time-out).

Example (Impact of the value selection strategy)

Consider `var 1..2: x`, `var 1..4: y`, `var 1..6: z`, domain consistency for `x*y=z`, `x!=y`, `x!=z`, and `y!=z`, smallest-domain variable selection, and depth-first search:

- If the values are selected by increasing order, then 6 CP nodes are explored before finding a solution.
- If the values are selected by decreasing order, then only 2 CP nodes (the root and a leaf) are explored before finding the solution, without backtracking.

Definition (Best-First Principle)

Pick a value that is most likely to lead to a solution.
Ideally pick a value that participates in solutions.

Example (Impact of the value selection strategy)

(Continued from slide 17)

Finding the first solution to `101`-queens with Gecode (CP):

search	# nodes	seconds
default (no annotation)	348,193	5.5
`first_fail, indomain_min`	348,193	5.6
`first_fail, indomain`	323,275	5.3
`first_fail, indomain_median`	96	0.1

UPPSALA UNIVERSITET

Motivation for First-Fail and Best-First[1]

	Finding a solution	Detecting unsatisfiability
Variable selection	Must consider all the remaining variables	Need not consider all the remaining variables: ☞ try and detect unsatisfiability a.s.a.p.
Value selection	Need not consider all the values: ☞ try and find a solution a.s.a.p.	Must consider all the values

[1] Based on material by Yves Deville and Pascal Van Hentenryck

Definition (Integer Brancher)

A brancher `int_search(`$\mathtt{X}$`,`ϕ`,`ψ`,complete)` selects an unfixed variable in the array $\mathtt{X}$ of integer decision variables, using as variable selection strategy ϕ one of the following:

- `input_order`: select the next variable by order in $\mathtt{X}$
- `first_fail`: select a variable with smallest domain
- `smallest`: select a variable with smallest minimum
- `largest`: select a variable with largest maximum
- `occurrence`: select a variable involved in the largest number of active propagators
- `most_constrained`: use `first_fail` and break ties with `occurrence`
- `max_regret`: select a variable with the largest difference between its two smallest domain values
- ... (see the MiniZinc documentation)

Ties are broken by the order in $\mathtt{X}$. (Continued on next slide)

Definition (Integer Brancher, continued)

Then, for the chosen variable, say v, the brancher selects values in $\mathrm{dom}(v) = \{d_1, \ldots, d_n\}$, with $n \geq 2 \wedge d_1 < \cdots < d_n$, and builds guesses, which are constraints, using as value selection strategy ψ one of the following:

- `indomain`: branch left-to-right on $v = d_1, \ldots, v = d_n$
- `indomain_min`: branch left on $v = d_1$, right on $v \neq d_1$
- `indomain_middle`: select d_i nearest $\dot{m} = \lfloor (d_1 + d_n)/2 \rfloor$ and branch left on $v = d_i$, right on $v \neq d_i$
- `indomain_median`: select median $d_i = d_{\lfloor (n+1)/2 \rfloor}$ and branch left on $v = d_i$, right on $v \neq d_i$
- `indomain_split`: branch left on $v \leq \dot{m}$, right $v > \dot{m}$
- `indomain_reverse_split`: left $v > \dot{m}$, right $v \leq \dot{m}$
- `outdomain_random`: select a random value d_i and branch left on $v \neq d_i$, right on $v = d_i$
- ... (see the MiniZinc documentation)

Definition (Boolean Brancher)

A brancher `bool_search(X,`ϕ`,`ψ`,complete)` selects an unfixed variable in the array `X` of Boolean decision variables, using variable selection strategy ϕ and value selection strategy ψ, with the same choices as for integer variables, under the convention `false < true`.

Definition (Chaining of Branchers)

A brancher `seq_search([`$\beta_1, \ldots, \beta_n$`])` chains branchers $\beta_1, \ldots, \beta_n$: when brancher β_i is finished, branch with β_{i+1}.

Careful: A brancher annotation goes between the `solve` and `satisfy`, `minimize`, or `maximize` keywords, and it is ignored elsewhere. See the example on slide 37.

Definition

A set (decision) variable takes an integer set as value, and has a set of integer sets as domain. For its domain to be finite, a set variable must be a subset of a finite set Σ.

Integers are totally ordered, but sets are partially ordered: propagation for set variables is harder. Also, set domains can get huge: $\mathcal{O}(2^{|\Sigma|})$. A trade-off is to over-approximate the domain of a set variable S by a pair $\langle \ell, u \rangle$ of finite sets, denoting the set of all sets σ such that $\ell \subseteq \sigma \subseteq u \subseteq \Sigma$:

- ℓ is the current set of mandatory elements of S;
- $u \setminus \ell$ is the current set of optional elements of S.

Example

The domain of a set var represented as $\langle \{1\}, \{1,2,3,4\} \rangle$ has the sets $\{1\}$, $\{1,2\}$, $\{1,3\}$, $\{1,4\}$, $\{1,2,3\}$, $\{1,2,4\}$, $\{1,3,4\}$, and $\{1,2,3,4\}$. Removing $\{1,2,3\}$ is impossible!

Definition (Set Brancher)

A brancher `set_search(`$\mathtt{x}$`,`ϕ`,`ψ`,complete)` selects an unfixed variable $S \doteq \langle \ell, u \rangle$ in the array $\mathtt{x}$ of set variables, using a variable selection strategy ϕ on slide 21:

- `first_fail`: select a variable with smallest $|u \setminus \ell|$
- `smallest`: select a variable with smallest $\min(u \setminus \ell)$
- ... (see the MiniZinc documentation)

Then, for the chosen variable, say $S \doteq \langle \ell, u \rangle$, it selects an element in $u \setminus \ell = \{d_1, \ldots, d_n\}$, with $d_1 < \cdots < d_n$, and adds guesses using a value selection strategy ψ on slide 22:

- `indomain_min`: branch left on $d_1 \in S$, right on $d_1 \notin S$
- `outdomain_max`: left on $d_n \notin S$, right on $d_n \in S$
- `outdomain_median`: select median $d_i = d_{\lfloor (n+1)/2 \rfloor}$ and branch left on $d_i \notin S$, right on $d_i \in S$
- ... (see the MiniZinc documentation)

Designing Search Strategies

Problem-specific strategies:

Beside general principles (first-fail and best-first), there are often good strategies that can be designed using problem-specific knowledge. In MiniZinc, it is often easy to express such strategies in terms of problem-specific concepts.

Interaction with symmetry-breaking constraints:

For higher solving speed, do not pick a value selection strategy that drives the search towards solutions ruled out by the symmetry-breaking constraints.

Example

For `a + b + c = 38`, with all variables in `1..19`, and `symmetry_breaking_constraint(a<b /\ b<c)`, do not use

```
int_search([a,b,c],input_order,indomain_max,complete).
```

UPPSALA UNIVERSITET

Outline

1. Annotations

2. Inference Annotations for CP & LCG

3. Search Annotations for CP & LCG

4. Case Studies

- Balanced Incomplete Block Design
- Warehouse Location
- Sport Scheduling

UPPSALA UNIVERSITET

Outline

Agricultural experiment design, AED

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	1	1	1	0	0	0	0
corn	1	0	0	1	1	0	0
millet	1	0	0	0	0	1	1
oats	0	1	0	1	0	1	0
rye	0	1	0	0	1	0	1
spelt	0	0	1	1	0	0	1
wheat	0	0	1	0	1	1	0

Constraints to be satisfied:

1. Equal growth load: Every plot grows 3 grains.
2. Equal sample size: Every grain is grown in 3 plots.
3. Balance: Every grain pair is grown in 1 common plot.

Instance: 7 plots, 7 grains, 3 grains/plot, 3 plots/grain, balance 1.

General term: balanced incomplete block design (BIBD).

The following constraints (of Topic 5: Symmetry) break the full row and column symmetries, but not their compositions:

```
15 constraint symmetry_breaking_constraint(
16  forall(v in Varieties diff {max(Varieties)})
17   (lex_greater(BIBD[v,..],BIBD[v+1,..])));
18 constraint symmetry_breaking_constraint(
19  forall(b in Blocks diff {max(Blocks)})
20   (lex_greatereq(BIBD[..,b],BIBD[..,b+1])));
```

The use of `lex_greatereq` (as opposed to `lex_lesseq`, say) is justified by the following search strategy:

- All `BIBD[v,b]` variables have the same `0..1` domain, so the first-fail principle cannot distinguish between them: let us fill the `BIBD` incidence matrix in input order (left-to-right in each row, and top-down across rows).
- Since typically fewer `1`s than `0`s occur in a BIBD, the best-first principle suggests trying `1` before `0`:

```
22 ::int_search(array1d(BIBD),input_order,indomain_max,complete)
```

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

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

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

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

$$\texttt{maintCost} = 30$$

$\texttt{Capacity} =$

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: Variables (Reminder)

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

`Supplier` =

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

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

Redundant decision variables:

`Open` =

Berlin	London	Ankara	Paris	Rome
$\in$ `0..1`	$\in$ `0..1`	$\in$ `0..1`	$\in$ `0..1`	$\in$ `0..1`

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

WLP Model 1: Search Annotation

The capacity constraint and the channelling constraint of `Supplier` with `Open` are as in Topic 6: Case Studies.

Let the new, redundant variable `Cost[s]` represent the actual supply cost for shop `s`, with channelling constraint:

```
forall(s in Shops)(Cost[s]=SupplyCost[s,Supplier[s]])
```

The objective becomes:

```
minimize maintCost * sum(Open) + sum(Cost)
```

For shop `s`, let dom(`Cost[s]`) $= \{d_1, d_2, \ldots, d_n\}$, with $n \geq 2 \wedge d_1 < d_2 < \cdots < d_n$: the regret of shop `s` is $d_2 - d_1$, that is the difference in supply cost between its currently cheapest and second-cheapest suppliers.

The maximal-regret strategy recommends:

- **Variable selection:**
  Choose a decision variable `Cost[s]`
  such that the shop `s` currently has the maximal regret.
- **Value selection and guesses:**
  Choose the smallest value d in dom(`Cost[s]`).
  Branch left on `Cost[s]` $= d$, right on `Cost[s]` $\neq d$.

The `Supplier[s]` decision variables are then branched on by increasing order of `s` and by increasing value.
This step is necessary only if, for some shop `s`, some values in `SupplyCost[s,..]` are equal.

Upon one-way channelling from `Supplier` to `Open`, the `Open[w]` decision variables are then branched on by increasing order of `w` and by increasing value, in order to set any still unassigned variables to `0`.

This search strategy is expressed in MiniZinc as follows:

```
1 solve
2   :: seq_search([
3         int_search(Cost,max_regret,indomain_min,complete),
4         int_search(Supplier,input_order,indomain_min,complete),
5         int_search(Open,input_order,indomain_min,complete)
6   ])
7   minimize maintCost * sum(Open) + sum(Cost)
```

Objective values, upon the three seen ways of channelling, within 35 seconds by Gecode (CP) on a MacBook-Air laptop, on a hard instance with 16 warehouses of capacity 4 supplying 50 shops, of minimal cost at most 1,190,733:

	Model 1		Model 2
search	one-way	two-way	none
default (no annotation)	none	1,869,494	1,864,913
first-fail on `Supplier`	1,520,326	1,524,034	1,524,034
first-fail on `Cost`	1,218,079	1,223,704	1,218,079
max-regret on `Cost`	1,193,637	1,198,276	1,193,637

UPPSALA UNIVERSITET

Outline

The Sport Scheduling Problem (SSP)

Find schedule in $\texttt{Periods} \times \texttt{Weeks} \rightarrow \texttt{Teams} \times \texttt{Teams}$ 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 $\texttt{n=8}$

	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 schedule in $\texttt{Periods} \times \texttt{Weeks} \rightarrow \texttt{Teams} \times \texttt{Teams}$ 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 $\texttt{n=8}$,
with a dummy week $\texttt{n}$ 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: Variables (reminder)

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.

`Team` =

	Wk 1		⋯		Wk $n-1$		Wk n	
	1	2	⋯	⋯	1	2	1	2
P 1	$\in$ T	$\in$ T	⋯	⋯	$\in$ T	$\in$ T	$\in$ T	$\in$ T
⋮	⋮	⋮	⋱	⋱	⋮	⋮	⋮	⋮
P $n/2$	$\in$ T	$\in$ T	⋯	⋯	$\in$ T	$\in$ T	$\in$ T	$\in$ T

`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: More Variables (reminder)

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

$\texttt{Game} =$

	Week 1	$\cdots$	Week $n-1$
Period 1	$\in$ Games	$\cdots$	$\in$ Games
$\vdots$	$\vdots$	$\ddots$	$\vdots$
Period $n/2$	$\in$ Games	$\cdots$	$\in$ Games

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

SSP Model 1: Channelling Constraint

Channelling constraint (reminder):

```
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`.

If a CP or LCG solver cannot enforce domain consistency on linear equality, even when `::domain` is used, then precompute a `table` constraint, say for `n=4`:

```
forall(p in Periods, w in Weeks)
  (table([Team[p,w,1],Team[p,w,2],Game[p,w]],
   [|1,2,6|1,3,7|1,4,8|2,3,11|2,4,12|3,4,16|])
```

SSP Model 1: Search Annotation

It suffices to follow the first-fail principle:

- **Variable selection:**
  Choose a decision variable `Game[p,w]` with the currently smallest domain.
- **Value selection and guesses:**
  Choose the smallest value d in dom(`Game[p,w]`).
  Branch left on `Game[p,w]` $= d$
  and right on `Game[p,w]` $\neq d$.

The redundant `Team[p,w,s]` decision variables need not be considered in the decision variable selection, as they take their values through the 2-way channelling constraint if the latter is propagated to domain consistency.

This search strategy is expressed in MiniZinc as follows:

```
:: int_search(Game,first_fail,indomain_min,complete)
```

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

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

Determine the period of each game, but not its week!

Search strategy:

Choose games for the first period across all the weeks, then for the first week across all the remaining periods, then for the second period across all the remaining weeks, then for the second week across all the remaining periods, etc.

Interested in More Details?

For more details on WLP & SSP and their strategies, 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.